AFIT/GCE/ENG/92D-03



# A VHDL Interpreter for Model-Based Diagnoses

THESIS

David Robert Griffin
Captain, USAF

AFIT/GCE/ENG/92D-03

AFIT/GCE/ENG/92D-03

# A VHDL Interpreter for Model-Based Diagnoses

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering



David Robert Griffin, B.S.E.E.

Captain, USAF

December, 1992

## *Preface*

The program used in this research was named after Susan Calvin, a fictional character created by Isaac Asimov. Calvin was the head robot-psychologist of U.S. Robotics and Mechanical Men. In several of Asimov's stories, Calvin diagnosed the robot's failures using her knowledge of the rules of robotics and the symptoms that the robots displayed.

I would like to thank the members of my thesis committee for the help they gave during my research at the Air Force Institute of Technology. I would especially like to thank Major Gregory Gunsch, who helped me through many rough areas. I would also like to thank my parents, Charles and Bobbie Griffin, who kept my spirits up during these last 18 months. Finally, I would like to thank Chesapeake and Taflina, who coiled their furry tails around my keyboard during long sessions with the computer.

David Robert Griffin

*Table of Contents*

## *List of Figures*

*Abstract*

Model-based reasoning permits diagnostic applications to be written without waiting for someone to become an "expert" of the system. For model-based diagnostics, there must be a model to reason from. This thesis explores using a VHDL description of the system as that model. A system based around a VHDL interpreter was written specifically for a model-based diagnostic algorithm. Currently, the diagnostic system uses an algorithm by Dries. This algorithm was derived from Scarl's Full Consistency Algorithm. The system was designed to be modular so that different diagnostic techniques could be implemented. It is divided into three parts: a VHDL parser, a VHDL interpreter, and a set of routines to implement Dries' Diagnose algorithm. The system can find stuck-at faults on combinatorial digital circuits.

# A VHDL Interpreter for Model-Based Diagnoses

## *I. Introduction*

### *1.1 Background*

*1.1.1 Model-Based Diagnostics* Several efforts at diagnosis using artificial intelligence have been based around production systems. When the production system is questioned about areas that it has been programmed, the system can give an answer. However, production systems have several limitations. The first is the expert: there must be someone who knows how the unit being diagnosed works. This person must be located, and a knowledge engineer must get the knowledge of the unit being diagnosed. For new products, there may not even be an expert. Even after the expert is found, the expert may not be able to explain his knowledge in enough detail for the knowledge engineer to code into the production system.

Assuming the knowledge engineer locates the expert, and that the knowledge engineer can translate the expert's knowledge into the rules for the production system, that knowledge still has limitations. If the production system is presented with a problem for which it has not been implicitly or explicitly programmed, it is unable to give an answer. Since the production system lacks deep knowledge of the unit being diagnosed, the production system cannot reason beyond the symptoms it was programmed to recognize. With no knowledge on how the system works, the production system cannot reason beyond the rules that are programmed.

Another problem with production systems is their rigidity. Once the production system has been programmed to diagnose one kind of unit, the production system can only diagnose that one type of unit. If modifications are made, or if the unit is redesigned, the production system may not

work for the new model. Sometimes the production system can be updated, but this could require re-consulting the human expert on the system.

Model-based reasoning attempts to overcome the limitations of production systems by attempting to "understand" how the unit being diagnosed works. By comparing the model with the faulty unit, and by knowing relationships between the components of the unit, model-based reasoning attempts to find out which component or components are at fault. Production systems attempt to find the faulty part by checking programmed knowledge that ties symptoms to specific problems. Model-based reasoning uses knowledge about the interconnections of the parts, as well as the knowledge on how each part is supposed to work, to come up with a diagnosis. Since the model-based reasoning system uses a model of the unit being tested, there is no need to consult an expert about every possible fault that can happen.

Compared to diagnostic systems based on production systems, model-based reasoning systems are a recent development. At AFIT, there have been a few thesis efforts dealing with model-based reasoning. In 1990, Kenneth Cohen described a method for diagnosing electronic modules, and implemented an assumption-based truth maintenance system. This is one of the components needed in a model-based diagnostic system. Cohen's method is described in greater detail in section 2.3.(1)

Also during 1990, Flight Lieutenant Ralph Dries developed a system for detecting anomalies in a satellite's pitch and velocity control subsystems. A model of the satellite's subsystems was modeled in Scheme/SCOOPS. The diagnostic system used model-based reasoning to find faults by comparing this model with a simulation of the real system. Dries' approach is discussed in section 2.5.(6)

In 1988, Captain James Skinner used a combination of a production system and a model-based system to diagnose the Dual Miniature Inertial Navigation system. In his Blended Diagnostic System (BDS), the system uses production system techniques to try to find the fault. If unsuc-

cessful, the BDS tries deep model-based reasoning on the sub-unit that appears to be at fault. (14)

Outside AFIT there have also been several efforts dealing with diagnostic systems based on model-based reasoning. These include approaches based on Reiter's Algorithm and abductive reasoning, described in sections 2.6 and 2.7. However, there has not been much done on describing the model for the reasoning system.

*1.1.2 VHDL* VHDL (VHSIC Hardware Description Language) is a hardware description language for designing Very High Speed Integrated Circuit (VHSIC) chips. That is, VHDL is a software system that simulates a hardware system. A designer can use VHDL to specify the operation of the VHSIC circuits. Once the designer has the overall behavior of the circuit specified, the individual components can be broken down into a more detailed design. This can go all the way down to the individual gate level. As each subcomponent is designed, its behavior can be simulated and matched against the specified behavior.

VHDL uses three main models: a timing model, a structural model, and a behavior model. A VHDL system simulates each component in parallel. The timing model allows the VHDL system to simulate each component in operating in parallel on non-parallel machines. The timing model is event driven: Each process (component) schedules the transactions. The timing model allows VHDL simulations to give the same results on different machines.

The structural model decomposes the complete system being simulated into various subsystems. This creates a hierarchy of subsystems, where simple subsystems are connected into higher-level subsystems. Ultimately, the higher level subsystems are connected to form the complete system. Each subcomponent is the equivalent of a "black box," each with a specified set of inputs and outputs.

The behavioral model describes how each subsystem works. This is one of the most complex parts of the language. The behavioral description can be as simple as single operation, or can be complex, with looping and conditional operations.

VHDL is a powerful language for hardware description. Although it is primarily used for digital design, VHDL has analog functions that should allow it to simulate non-digital systems. VHDL has been standardized by the IEEE (IEEE-1076). It is also accepted by the U.S. Government as a standard for VHSIC design(5:4).

A model-based reasoning system requires a model. Determining the model of the system to be tested can be a difficult task. This research explored using VHDL for specifying the model.

### *1.2 Problem*

One of the problems of model-based diagnostics is creating a model of the system to be tested. This research effort is use a VHDL description of a circuit as the model. This avoids the need to create an additional model for the diagnostic system.

### *1.3 Scope*

The goal of this research was to design and implement a diagnostic system, named Calvin, that used VHDL to describe the test system. Calvin was designed so that it can be extended to handle diagnostic algorithms.

The following limitations applied to this research:

- Tested systems were feed-forward combinatorial digital circuits.
- A subset of the VHDL language was implemented. The subset was enough to describe the above circuits.
- The VHDL description of the circuits accurately describe the operation of the circuit.

- Time-sensitive behavior, such as memory, was not explored.
- Components were composed of only lower-level subcomponents, or boolean algebra descriptions.
- Only the following faults were simulated:
  - Output stuck high
  - Output stuck low
  - Input stuck high
  - Input stuck low.
- If an input was stuck high or low, it was assumed to be disconnected from the rest of the system.
- The tested system had at most one fault.

## *1.4 Approach*

There were three main areas in this research effort: Parsing the VHDL language, simulating the circuit, and interfacing a diagnostic method to Calvin. There also must be some means for testing Calvin.

*1.4.1 VHDL Parser* The parser took an input file and represented it internally. A VHDL grammar written for the GNU Bison compiler-compiler was used as the skeleton for the parser. This implementation is described in Chapter III. Only a subset of the VHDL language was used for Calvin; unimplemented VHDL constructs were ignored.

Objective: Be able to read and parse the VHDL source code files for the test circuits.

*1.4.2 VHDL Simulator* To perform model-based reasoning, there must be some way to exercise the model. In this research effort, VHDL was the model; therefore, there had to be a way

to simulate the VHDL source code. Although there are already VHDL simulators in existence, this system was designed to allow easy interfacing to diagnostic routines.

VHDL is a complex language; a full implementation of the language is well beyond the scope of this research. A description of the VHDL subset is in Appendix A. Chapter III describes the implementation of the simulator.

Objective: Be able to simulate test circuits and generate expected values of the VHDL signals.

*1.4.3 Diagnostic Routines* To perform the diagnosis on the test circuits, a set of routines was interfaced with the VHDL simulator. The goal was to make both the simulator and the diagnosis routines loosely coupled. Chapter II contains previous research into model-based diagnosis.

The implementation of the diagnostic routines is discussed in Chapter III, along with a description of how they were integrated with the VHDL portion of Calvin. Chapter IV contains extensions that were explored, but not implemented as of this time.

Objective: Select and implement a model-based diagnostic strategy, and interface it with a VHDL simulator.

*1.4.4 Selection of Test Circuits* To test Calvin, there needed to be a set of sample circuits. Since this thesis investigation was implementing a subset of the source language, circuits were selected that only used the subset. These circuits are discussed in Chapter IV.

Objectives: Create a sample of test circuits for Calvin to analyze.

## *1.5 Thesis Overview*

The next chapter reviews some model-based diagnostic methods that have been used by other researchers, including past AFIT thesis efforts. Chapter III describes the implementation of the VHDL parser, simulator, and diagnostic routines. The results of the implementation of Calvin are discussed in Chapter IV, along with a discussion of ways to enhance the diagnostic routines.

Finally, Chapter V will state the conclusions found from this research, along with recommendations for future efforts.

# *II. Literature Review*

## *2.1 Introduction*

There are many different ways for performing model-based diagnostics. This chapter reviews algorithms that previous researchers developed. Included are descriptions of model-based research done by past students of the Air Force Institute of Technology. This chapter concludes with a review of some considerations that development of a model must address.

## *2.2 Reasoning from First Principles*

One method of model-based reasoning was developed by Randall Davis in 1984. In it he discussed problems with previous efforts at troubleshooting systems. Davis proposed to solve the problems by developing a system that reasons from first principles, using knowledge of the structure and behavior of the system (4:347).

To reason about structure and behavior requires ways of representing both. Davis based his structure description on three ideas: ***modules***, ***ports***, and ***terminals*** (4:352). Modules were the black boxes that made up the system. Information flowed in and out of the modules through ports. Each port had two or more terminals: one on the outside of the module, and one or more on the inside. Modules were connected by superimposing their terminals together. There were no separate entities for dealing with wires; if a wire was explicitly modeled, it was simply another module. The module descriptions were hierarchical. A module may be decomposed into submodules.

The behavior of the system being tested also must be modeled. To support Davis' technique, the behavior of the modules was described by a combination of simulation rules and inference rules. The simulation rules described the output of the module as a function of its inputs. Inference rules inferred the possible values of one input as a function of the rest of the module's inputs and its output. Simulation rules represented the flow of behavior, while inference rules represented the

| | | | | | |
|---|---|---|---|---|---|
| to get | sum | from | (input-1 input-2) | do (+ input-1 | input-2) |
| to get | input-1 | from | (sum input-2) | do (- sum | input-2) |
| to get | input-2 | from | (sum input-1) | do (- sum | input-1) |

Figure 1. Behavioral Description of an Adder Module
(4:357)

flow of inference (4:358). An example of a behavioral description of an adder module is shown as Figure 1. The first line is the simulation rule. The other two are inference rules.

Davis described the traditional approach to troubleshooting as a theory of test generation, not diagnosis (4:360). The test generation approach was to hypothesize possible faults, and then determine a set of input values that would logically detect that fault. This approach did not provide any insight on determining which component to consider next. The traditional approach also required all faults to be explicitly enumerated. Other faults, such as those caused by solder bridging two points in the circuit, could not be diagnosed.

To avoid the problems with past techniques, Davis proposed the use of discrepancy detection. Instead of hypothesizing faults, this technique looked for observed values that were different from the simulated values. Misbehavior was then defined as anything that wasn't correct (4:362). A dependency network contained all components that could influence the incorrect output. The components in this network were the suspects that needed to be checked. Each component was checked by seeing if there was any assignment of values to its ports that could produce the observed state of the entire system. Since to do this required that the behavior of the suspected component to be temporarily ignored, Davis called this procedure ***constraint suspension***. If a consistent set of values could be assigned to all the ports in the system, the component was kept as a possible suspect. However, if there was no way to assign values that were consistent with the known outputs, the suspect component alone could not cause the observed behavior (4:364).

### *2.3 Assumption-based Truth Maintenance System*

In 1990, AFIT student Kenneth Cohen developed a model-based reasoning system that was based on Davis' reasoning from first principles. In his thesis, Kenneth Cohen created a model-based reasoning system that consisted of three parts: a model-maker module, a diagnostic engine module, and a truth maintenance system module. The model-maker module was used to model the system to be diagnosed. The model-maker had to be able to generate "correct" behavior for the system. The diagnostic engine module compared actual observations with those generated by the model-maker module. If the diagnostic engine detected a discrepancy, the diagnostic engine attempted to find the cause of the problem. Using constraint suspension, the diagnostic engine tested sets of components to see if a set might cause the observed symptoms. (1:17-27)

The main thrust of Cohen's thesis was the truth maintenance system module, called the Assumption-based Truth Maintenance System (ATMS). ATMS was a method for keeping track of assumptions for a model-based diagnostic system. It had three roles: it "remembered" previously made inferences, it allowed base assumptions to be made, and it maintained an environment free of contradictions. (1:30)

By remembering inferences, ATMS reduced computation. If a component's value had been calculated once, that value would not have to be recomputed for the same inputs (1:30). The second role was to allow base assumptions to be made. This allowed other beliefs to be reasoned (1:30). ATMS also maintained contradiction-free environments. Assumptions were usually of the form "if there is no reason to believe $\neg P$ then believe $P$" (1:31). ATMS retracted any assumptions that conflicted.

Since Cohen was concentrating on the ATMS, he did not implement the model-maker module. Instead, the model of the test system was hard-coded in Lisp. (1:36)

### *2.4 Full Consistency Algorithm*

A different approach to model-based reasoning was developed by Scarl, Jamieson, and Delaune. Scarl's paper described a prototype system for monitoring a liquid-oxygen expert system. The diagnostic system, called LES, determined faults from sensor data using knowledge of structure and function of the liquid oxygen system. (13:360-361)

LES had to have a model of the system that was to be tested. This model was a network of objects, each representing a subcomponent of the system that was to be tested. An object description contained the type of object. Two types of objects,*commands* and *sensors*, contain measured or assigned values and tolerances. The LES algorithm also required three other descriptors: the *source*, the *source-path*, and the *status*. The *source* pointed to the source of this object's value. The *source-path* determined if this object was connected to the object specified by the *source*. This value was a boolean. For digital objects, these descriptors were enough to describe the object. If the object was an analog object, a *status* field was required to determine the state of the object when the *source-path* field was on. (13:361-362)

The fields in the object's descriptors contained expressions that determined the value of the object. These expressions contained the names of other objects in the system being tested. When calculating the value of the object, the names of other objects were replaced with the value of the other object that was named. (13:362)

Objects were divided into three categories: *commands*, *components*, and *sensors*. The *commands* entered values into the system. *Components* took values at their input, and generated an output value. *Sensors* only measured a value. They could not modify any other object in the system. Information in the model only flowed in one direction, from outputs of objects into the inputs of other objects. Each object was assumed to have only one output, which could be connected to the inputs of several objects. (13:362)

The LES tested for faults whenever a command or sensor value changed. When a sensor reported a value, it was checked by computing its expected value. LES did this by evaluating the *source-path* or *status* expressions. Since these expressions contained the names of other objects, these were also evaluated. The objects were recursively evaluated until the values stored in the command objects were reached. If the observed value for the sensor was within the range of the calculated value for that sensor, the sensor was labeled as consistent. If the observed and calculated values did not match, this sensor was labeled discrepant, and LES invoked the diagnoser. (13:364)

When a command value was changed, all sensors affected by this command were checked. The LES compared observed values with computed values, and were labeled accordingly. The first sensor that didn't match its calculated value caused the diagnoser to be invoked. (13:364)

During diagnosis, objects were labeled *innocent*, *culprits*, or *suspect*. *Innocent* objects were those objects that could not cause the faulty value of the sensor. *Culprits* were those objects which LES had decided could cause the fault sensor value. If an object was not *innocent* or a *culprit*, it was labeled a *suspect*. The sensor that didn't match its calculated value was labeled the Original Discrepancy, or OD. This sensor also could be a culprit, a suspect, or innocent. (13:362)

Scarl's Full Consistency Algorithm for finding possible faults is as follows:

1. Pick a system object and label it as a suspect. Only objects that are upstream from the OD are considered. LES picks suspect objects by keeping track of objects visited while calculating the expected value of the OD. Since the sensor itself may be malfunctioning, it also will be picked as a suspect.
2. Hypothesize a faulty state for the suspect object. Since the correct value of the object can be calculated, any other value for the object represents a faulty state. The faulty state is not picked randomly. Instead, the expressions for *source-path* or *status* are inverted. Fault states are determined based on these inverted expressions.

3. Assume the faulty state for the object. Simulate the system and determine values for all the sensors.

4. Compare the simulated values with those of the actual system. If the simulated values are consistent with those actually measured, the hypothesized fault is one possible explanation of the original faulty OD. If the simulated values are not consistent with the measured values, the hypothesized fault is ruled out.

5. If the simulated and measured values are not consistent, and there are more possible faults, loop back to step 2 and hypothesize a different fault. If there are no more possible faults with the object, the object is labeled innocent.

(13:364)

This algorithm requires several assumptions. The system must not have any feedback loops. Only one fault may occur at a time. LES could handle multiple failures, but only if each failure could be diagnosed before the next one occurred. The equations that describe the objects contained wild card values. LES used these to represent indeterminate states. These were used to switch out objects LES had determined to be faulty. The algorithm also assumed that the sensor polling cycle was shorter than the length of the faulty behavior. LES must be able to determine the faulty object before the faulty behavior changed. All objects in the system being tested could only have one output. Those objects that had more than one output were decomposed into sub-objects, each containing one output. Uncertainty was handled by using ranges. Tolerances were propagated backward as a range of possible values. An overlapping range would match an expected value to a measured value. (13:364)

### *2.5 Model-Based Reasoning in the Detection of Satellite Anomalies*

Flight Lieutenant Dries used Scarl's algorithm to develop a system for monitoring an Attitude and Velocity Control Subsystem (AVCS) of a geo-stationary satellite. Dries modified Scarl's

algorithm so that the diagnostic system did not have to invert the description of a component. Instead, Dries included in the behavior description of the component a list of possible faults. For each fault, the behavior description was modified so that the fault can be simulated.

Dries determined several characteristics of the language he would use to write the diagnostic system. Because the system model would be written in the language, and the model consisted of a network of objects, an object-oriented language would be required (6:72-73). Other considerations include a commonly used language that would produce efficient code. The language also should run on a personal computer. This would reduce development cost, since PC's are relatively inexpensive and are readily available (6:73). Languages he investigated include Smalltalk, Lisp with Flavors, C++ and Scheme with SCOOPS.

Originally, Dries tried to use the C++ language for the model and reasoner. Borland has a C++ compiler that is known for its efficient code generation and convenient user interface. However, his lack of familiarity with C, along with C's steep learning curve, prevented him from using C++. He then turned to SCOOPS to develop his system. Dries chose Scheme for its simplicity, symbol manipulation and fast prototyping ability (6:75). Both the model and the model-based reasoner were written using SCOOPS, an object-oriented extension to SCHEME. He concluded that C++ probably would be a better language for the final diagnostic system because of its object-oriented capabilities (6:75).

The components of the AVCS were modeled as SCOOPS objects. Using the object-oriented paradigm, Dries created a hierarchy of component classes. At the top of the hierarchy was a super-class called *component*. This class contained attributes that all the system objects have. These included a name, a status, a list of objects connected to this object's inputs, a list of objects connected to this object's output, and a state that can be transmitted to the objects in the output list. These attributes were implemented as *instvars*, or instance variables, in the SCOOPS object.

The status instvar was the same as the status descriptor described by Scarl. The source instvar of *component* corresponded to Scarl's source descriptor. (6:78-79)

The individual components were instances of the *component* class and its subclasses. SCOOPS automatically generates functions, known as "methods" for getting, setting, and initializing attribute values. A deposit-value method propagates the value of the object to other objects in the output-list.

Below the *component* class in the hierarchy was the *amplifier* class. Since the satellite system was primarily analog, most of the components were of the *amplifier* class and its subclasses. The *amplifier* class added other instvars that were needed by amplifiers. These included gain, limit and tolerance. The *amplifier* class also contained a list of possible faults, such as latch-up, high/low and zero. Dries wrote methods for this class that would simulate the operation of an amplifier, along with possible faults states. The specific components of the AVCS system were derived from the *amplifier* class. These added other instvars and modify the simulation methods. (6:80-82)

In Scarl's system, inputs arrived via *command* objects, while the outputs of the system were measured by *sensor* objects. Dries modeled the *command* object by deriving its class from the *component* class. Since a sensor could itself be faulty, it was modeled as a type of *amplifier* with a gain of 1 and a tolerance of .0001. The *sensor* class also contained a list of all objects upstream of itself. These were the possible objects that could affect this sensor. (6:94)

The pitch control channel of the AVCS was modeled by a network of instances of the various classes. For Dries' research, two networks were set up: one to represent the model, and one to represent the real subsystem. The SCOOPS "make-instance" instantiated each component. The input and output lists formed the interconnections of each network. (6:95-97)

As stated previously, Dries took Scarl's Full Consistency Algorithm, and modified it for his work. Dries' Reasoner Algorithm is described in Figure 2.

```
Find a discrepant sensor
If none found then
    No fault in circuit
Else
    Collect all components structurally upstream from
        discrepant sensor and put into suspect list
    Repeat for each suspect
        Repeat for each fault hypothesis
            Hypothesize a fault for the suspect
            Propagate change through the model
            Test all sensors for consistency
            If sensors consistent then
                Leave suspect in suspect list
            Else
                Clear hypothetical fault (not suspect)
        End-repeat faults
        If all faults are ruled out then
            Clear suspect
    End-repeat suspects
    If one suspect remains then
        Print out the culprit
    Else
        Print out the list of suspects remaining
```

Figure 2. Dries' Diagnose Algorithm
(6:98)

Because his algorithm was based on Scarl's algorithm, Dries' Reasoner algorithm was still subject to the same assumptions and limitations of Scarl's algorithm. One problem Dries encountered was that his pitch control system was a feedback loop. Since neither algorithm would work with a loop in the test system, the feedback loop had to be broken during the test phase (6:98). Instead of connecting the actual objects in the networks, Dries wrote a test-loop function that took the output of the system and injected it back into the input. When a fault was introduced into the system, the feedback loop in the model system was broken, and the diagnostic model invoked.

Another limitation of Scarl's algorithm was that objects could not be time dependent. Dries overcame this by modifying the time dependent objects so they were non-time dependent. Dries' system was still able to detect faults in those modified objects. (6:35)

When Dries ran his diagnostic program, the program was able to find almost all the faulty components he introduced. He concluded that this was a result of the model and the test system being exactly the same (6:109). Both the model and the pitch control system were made of the same SCOOPS objects. Dries stated that a better test of his system would be to use a more realistic real-world simulation, but at the same time use a computer model without time dependent objects (6:109).

### *2.6 Reiter's Algorithm with Enhancements*

This approach is an extension of Reiter's Algorithm. Where Reiter's algorithm was applied only to diagnosing digital circuits, this extension would cover systems that vary over time.

In Reiter's algorithm, called DIAGNOSE, a problem consisted of a set of system descriptions (SD), a set of the system's components (called COMP), and a list of observations of the system (OBS). A *diagnosis* was a subset of COMP that consists of faulty components. $SD \cup OBS$ (the description of the system, together with observations of the system) must be valid assuming all the components of the subset were faulty, and all components not members of the subset were not

faulty. A *conflict set* was a set of components such that assuming all the components in the set are normal is inconsistent with $SD \cup OBS$. (9:10)

The DIAGNOSE algorithm computed a set of all diagnoses by building a search tree, called a pruned HS-tree (heuristic search tree). Nodes of this tree were labeled with a conflict set, while the edges were set by a system component. Each node had a *path label*, which is the set of all edge labels from the root to that node. The algorithm required a consistency checking module, called TP. This module took the SD, OBS and a subset of COMP. It returned a conflict list, if one existed. Otherwise, it returned **null**. (9:10)

The HS-tree was set up by calling TP, passing it the entire COMP list. The root node was set to the returned conflict list. Then, for each element in that conflict list, a child node was created. It was connected to its parent node with an edge labeled by the element. The path label was then set to be the path from the root to that node. TP was called with the COMP list minus the elements in this path. The returned conflict list was the label for the child node. If 0 was returned, the child node was marked as *completed*. When the HS-tree was completed, the set of diagnoses was the set of all the path labels of the nodes marked completed. (9:10-11)

This paper described a way to extend Reiter's DIAGNOSE algorithm to handle time-varying systems, as well as continuous devices. The continuous device was broken into a set of components. Each component was described by one or more equations. It was assumed that the continuous device can be modeled by a component-connection model (9:11). Each constraint also could be localized to one particular component. This means that a constraint that was broken could be traced to one faulty component. SD was then set as the qualitative restraints of the system, and OBS as the set of qualitative states. The TP module was a constraint propagation module. When it was called with a subset of components, all restraints are removed except those related to the component subset. The propagator attempted to propagate the parameter values as much as possible. If a propagation was made by using a constraint, the constraint was marked as used.

If an inconsistency was detected, the TP module stopped and returned a list of components that have had one of their constraints marked. If the propagation halted with no inconsistencies, the TP module returned the empty set. This meant that all the components passed to the module were normal. (9:11-13)

To handle continuous devices, the DIAGNOSE algorithm was run using the initial set of observations. The conflict sets generated would have at least one faulty component. When a new observation was made, all the nodes marked completed were opened. The TP module was then called on each of these nodes, this time using the new observations. This was done until all the nodes had been processed. The final set of completed nodes became the new diagnosis set. (9:13-14)

### *2.7 Abductive Diagnostic Reasoning*

One problem with Reiter's Algorithm was that it might not pick the most "probable" faulty component. For example, assume a component can cause one symptom 95% of the time, and a second symptom 5% of the time. A different component could cause the second symptom 90% of the time, but would never show the first symptom. Reiter's algorithm would say that the only first component was faulty, even though it was more probable that both components were faulty (8:16). Abductive reasoning attempted to choose the most probable set of disorders.

Abductive reasoning was based on a causal network, a directed graph that describes the problem domain. The nodes were a set of events that include disorders, symptoms, and pathological states. The edges of the network were *direct causation events*. They connected an event that could directly cause another event with no known intervening events. A problem was stated by a list of observations, each being a node in the causal network. *Scenarios* were chains of causation events. *Causal explanations* were scenarios that hold true for the problem's observation set. Abductive reasoning attempts to find the causal explanation that was most probable. (8:17-18)

Each causal event was a given a probability when the causal network was constructed. The probability of a scenario was the product of the casual events that make up the scenario (8:18). The goal is to maximize the probability of a scenario that explained all observations. This became a variation of the Steiner Problem, a NP-Complete problem (8:19). The rest of this paper described an approach for reducing complexity to polynomial time to the number of nodes in the network.

### *2.8 Modeling Digital Circuits for Troubleshooting*

In this paper, Hamscher discussed problems with current models used in model based reasoning. He described a situation where a field engineer could diagnose and fix a circuit in ten minutes using only a few probes and swapping one chip. A model-based troubleshooting program took an entire day, and then concluded that any of 40 chips or 400 wires could be responsible for the problem (7:2). To overcome this problem Hamscher proposed incorporating knowledge on how the component could fail in the circuit model. He gave eight principles for modeling digital circuits. These are summarized here:

1. Components in the model should correspond to possible repairs. There is no point in determining which transistor in the chip is bad. If any part of the chip is bad, the whole chip will need to be replaced. This cuts down on the processing time spent in diagnosis. (7:6)
2. Model components should simplify behavioral abstraction. The only reason to represent a function in the model is to make the behavior prediction more efficient. If it is easier for the diagnostic system to reason about a group of components, group the components. (7:6)
3. Component behavior should represent features easy for the troubleshooter to observe. Some features are easier and more efficient to observe than others. (7:7)
4. Components whose behavior changes every time its inputs change should be represented in temporally coarse terms. More powerful representations take into account the function of the circuit over long periods of time. Hamscher gives as an example the number of mouse

increments per second determining the number of times an interrupt line would be asserted. (7:7)

5. A temporally coarse description that only describes some of the component's behavior is better than no description at all. An example would be a microprocessor chip interfaced to the mouse. The relationship between the motion of the mouse and the interrupts lines only holds true if the clock is running. The troubleshooting program can still use this behavior to find faults, though the entire function of the microprocessor is not simulated. (7:7)
6. Encapsulate sequential circuits into a single component. This cuts down on the number of behaviors that the troubleshooter must consider. The overall resulting behavior makes reasoning about the behavior more efficient than considering the various behaviors of the components of the circuit. (7:7)
7. If there are known likely failures in a component, represent the failure mode in the model. This can reduce the number of different diagnoses. (7:7)
8. If a component's misbehavior is much easier to model than the correct behavior, include the misbehavior in the component's model. If a component with complex behavior fails completely, then any partially correct behavior can make the component a much less likely suspect. Since a complete malfunction can usually be easily modeled, the troubleshooting system can efficiently detect the failures if they are explicitly modeled. (7:7)

## *2.9 Summary*

This chapter reviewed some current methods that researchers are using to perform model-based diagnostics. One recurring problem is how to model the system. This effort will use VHDL as a way of specifying a model. The implementation of such a system is discussed in the next chapter.

# *III. Implementation*

## *3.1 Overview*

*3.1.1 Introduction* For a model-based diagnostic system, there obviously must be some way to model the system to be diagnosed. One way of modeling the system is to use some form of hardware description language. This thesis will use VHDL as that language.

*3.1.2 The VHDL Language* VHDL was created primarily for the design and verification of large-scale integrated circuits (10:2). Its very name, VHSIC Hardware Description Language, signifies it as a language for describing (modeling) hardware. MIL-STD 454L requires that all new application-specific integrated circuits will have a VHDL description (5:4.5.1).

VHDL has several basic building blocks which my diagnostic system, named Calvin, must implement. These include *Entities*, *Architectures*, *Configurations*, and *Processes*. Others will be left for future research.

In VHDL the *Entity* is the most basic block in the design (10:3). The entity specifies what objects exist in the system. They are arranged in a hierarchy, with the top entity representing the system itself.

The *Architecture* describes how an entity behaves. There are two types of architectures: *Behavioral* and *Structural*. The *Behavioral* architecture describes how the entity behaves in terms of VHDL statements. The *Structural* architecture describes the architecture as interconnections of entities that make up the architecture. This creates the system hierarchy.

Since an entity may have more than one architecture, there must be some way to specify which architecture to use for the entity. VHDL uses the *Configuration* to bind the instances of an entity to a specific architecture.

The basic simulation block of the system is described by *Processes.* The behavioral architectures contain one or more processes to describe the operation of that architecture. All processes are assumed to be operating in parallel.

*3.1.3 Diagnose Algorithm* After determining how the test system was to be modeled, the next step was to determine if there was an error. When the diagnostic system detected an error, the diagnostic system needed some method of determining which component of the test system was at fault. Although there were many methods, some of which were discussed previously, this research used a method originated by Scarl and used in a previous thesis by Dries. This algorithm is shown in Figure 3. The method this research used had two advantages:

- No need to "invert" the VHDL source. Diagnostic methods such as those used by Davis required a way for determining the inputs of a system, given the values of the outputs.
- Fits in with a simulator-based modeling system. Possible faults in the individual components were determined before diagnoses. This allowed more work to be done before the actual diagnosis.

However, there were also a few disadvantages:

- No feedback. Neither Scarl's *full consistency* algorithm nor Dries' *diagnose* algorithm allowed the test circuit to have any feedback. This prevented state machines from being tested. Dries worked around this limitation by breaking the feedback loop during diagnostics.
- Limited fault detection. The set of faults the system will look for were predetermined before diagnosing the test circuit. If an unforeseen fault occurred, the system could not find it.
- Combinatorial explosion. The time needed to diagnose a system was dependent on the number of suspects. This in turn was dependent on the depth and branching factor of the test circuit. In an extreme case with only one sensor, every component would be suspect. This would result in testing every hypothesis for every component in the test circuit.

The system's diagnostic method should be easily updated when needed. This was done by a combination of object-oriented programming and loosely coupled modules. The algorithm was divided into two areas: *Generate* and *Test*.

*3.1.3.1 Generate* Calvin first generated the various hypotheses for each component. This was done during the parsing of the VHDL model. As an executable section of the model was parsed, the parsed data was sent to the hypothesis generator to determine what could go wrong. Each simulation component contained a set of instructions on how it was supposed to logically work. This section took the correct model and generated the faulty behaviors. Although currently this portion is only executed during parsing, it can be extended to be executed during simulation.

For this effort four common hypotheses for digital circuits were generated:

1. Input stuck high. This simulates the case where the input of a component always reads "high."
2. Input stuck low. This is where the input always reads "low."
3. Output stuck high. This simulates an output that is always high. This may seem to be the same as if an input connected to that output was stuck high. The difference is in this hypothesis *all* inputs connected to the output will be pulled high.
4. Output stuck low. This is where the output line always reads low.

*3.1.3.2 Test* This portion implements Dries' *Diagnose* algorithm. Figure 4 describes the algorithm used by Calvin.

After Calvin found an output that did not match its simulated value, the component attached to that output was placed in a suspect list. Then, Calvin works upstream, placing each component into the suspect list until the inputs were reached. Calvin did this by using the structure of the test circuit.

```
Find a discrepant sensor
If none found then
    No fault in circuit
Else
    Collect all components structurally upstream from
        discrepant sensor and put into suspect list
    Repeat for each suspect
        Repeat for each fault hypothesis
            Hypothesize a fault for the suspect
            Propagate change throughout the model
            Test all sensors for consistency
            If sensors consistent then
                Leave suspect in suspect list
            Else
                Clear hypothetical fault (not suspect)
        End-repeat faults
        If all faults are ruled out then
            Clear suspect
    End-repeat suspects
    If one suspect remains then
        Print out the culprit
    Else
        Print out the list of suspects remaining
```

Figure 3. Dries' Reasoner Algorithm
(6:98)

Calvin took each suspect from the list and tried to determine if it could cause the problem. It went through the hypotheses that were created during the initial parsing, and simulated the fault. The VHDL simulator was then rerun to see if that fault could account for the all the known output values. If so, that hypothesis was kept; otherwise it was thrown out. This process was repeated for the rest of the hypotheses and suspects.

### *3.2 The Calvin Diagnostic System*

A diagram of the Calvin system is shown in Figure 5. There are three main units in Calvin: the VHDL parser, the VHDL simulator, and the diagnostic routines. In Figure 5 the diagnostic routines are in the *Init*, *Hypothesis Generator* and *diagnostic* blocks. The code that controls the program flow is contained mostly in the modules **CALVIN** and **MAIN**.

```
Check sensors
    Collect suspects
    While more suspects
        While more hypotheses
            Hypothesize fault
            Re-simulate
            Compare all sensors with their simulated values
            If consistent
                Keep suspect
            Else
                Remove suspect
        End
    End
End
```

Figure 4. Calvin's Diagnostic Algorithm

The VHDL parser takes the source code and generates an internal representation of the circuit. While the parser is generating this representation, the data is sent to a hypothesis generator. This module creates code to simulate the errors that Calvin will check for during diagnosis. The simulator takes a current representation of the circuit and its inputs and determines what the outputs should be. This is done first to check the outputs of Calvin's model of the circuit against the measured outputs of the circuit. The simulator is also used to re-simulate the circuit after a fault is introduced. The diagnostic routines implement a version of Dries *Diagnose* algorithm. These routines call the simulator modules as needed for re-simulation.

The *Init* block in Figure 5 is further broken down in Figure 6. First, Calvin initializes the internal variables. Calvin then parses the command line to get the VHDL source file name, test inputs and outputs file name, and commands. An example is given in section 4.1.1. The VHDL source file is then opened and sent to the parser.

The VHDL parser takes a file containing the VHDL description of the system to be diagnosed and generates an internal representation of the system. During this parsing, the internal represen-

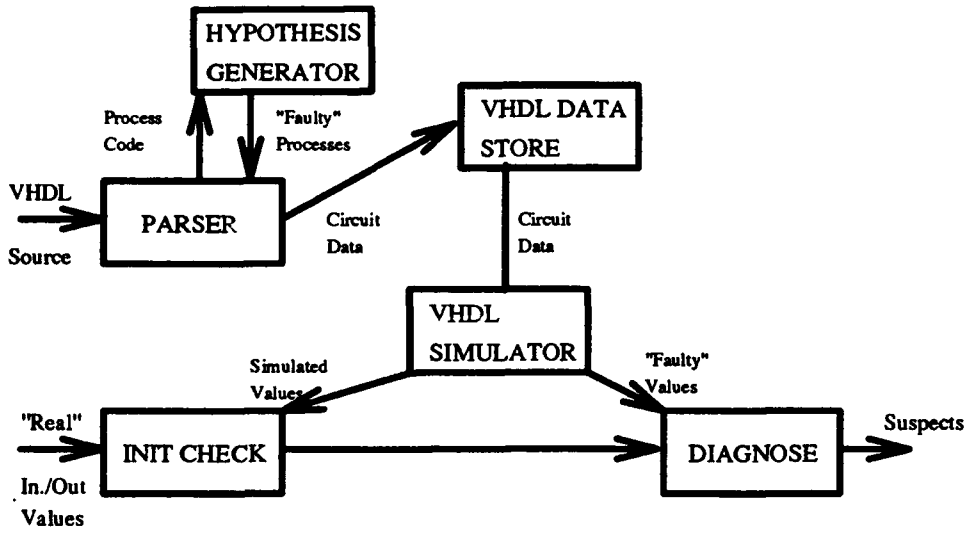


Figure 5. The Calvin System

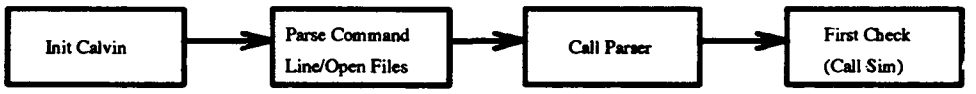


Figure 6. Calvin Initialization

tation is handed over to a diagnostic module to generate possible faults. This is the *Hypothesis Generator* shown in Figure 5.

The next step is to get the current inputs and outputs of the actual circuit (called by Scarl *commands* and *sensors*). These are contained in the test file specified on the command line. The input signals are set to these values. Control is given to the last block in Figure 6, *First Check*.

First, the VHDL simulator is called to generate the sensor values that Calvin expects for a correctly operating circuit. Calvin then compares these values with those reported in the test file. This is detailed in Figure 7. Calvin loops through each sensor and compares its simulated correct behavior with that of the "real world." If all the sensors match, Calvin decides that there are no faults and quits. Otherwise, the system calls the diagnostic routines. These routines implement a version of Dries' *Diagnose* algorithm, re-simulating the circuit as necessary. This is detailed in Figure 8. Possible suspects are collected based on which sensor does not match the simulated value. This routine is detailed in section 3.2.3.2. The suspects are collected in a queue. While there are

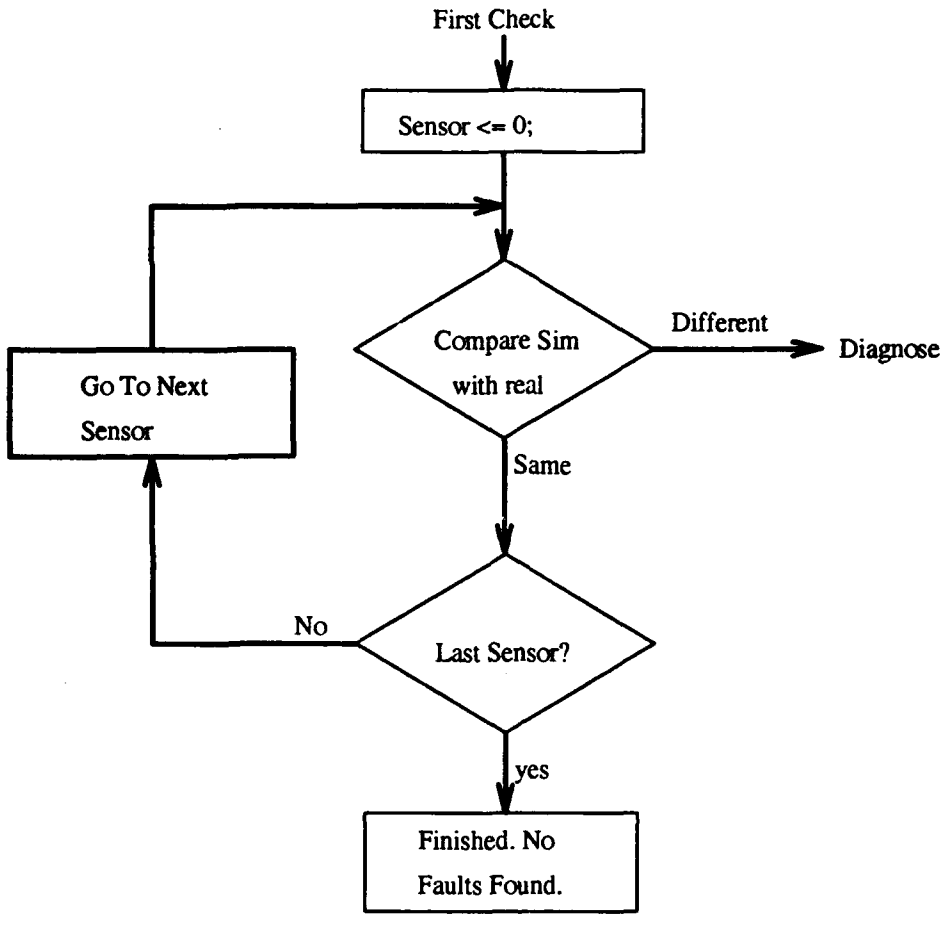


Figure 7. First Sensor Check

still suspects, Calvin takes one off the top. Calvin then runs through the possible hypotheses for this suspect. Each hypothesis is implemented in the simulated circuit, and the VHDL simulator is called. The newly simulated values are compared against those in the test file. If they match, Calvin reports that this is a possible fault in the circuit. Otherwise, Calvin rejects the hypothesis and selects the next. Once all the hypotheses are finished, Calvin goes to the next suspect in the queue. When all of the suspects are checked, Calvin quits.

*3.2.1 VHDL Parser* The parser section takes as its input the name of the file that contains the VHDL source code. Using a subset of the VHDL language, it generates a set of data structures

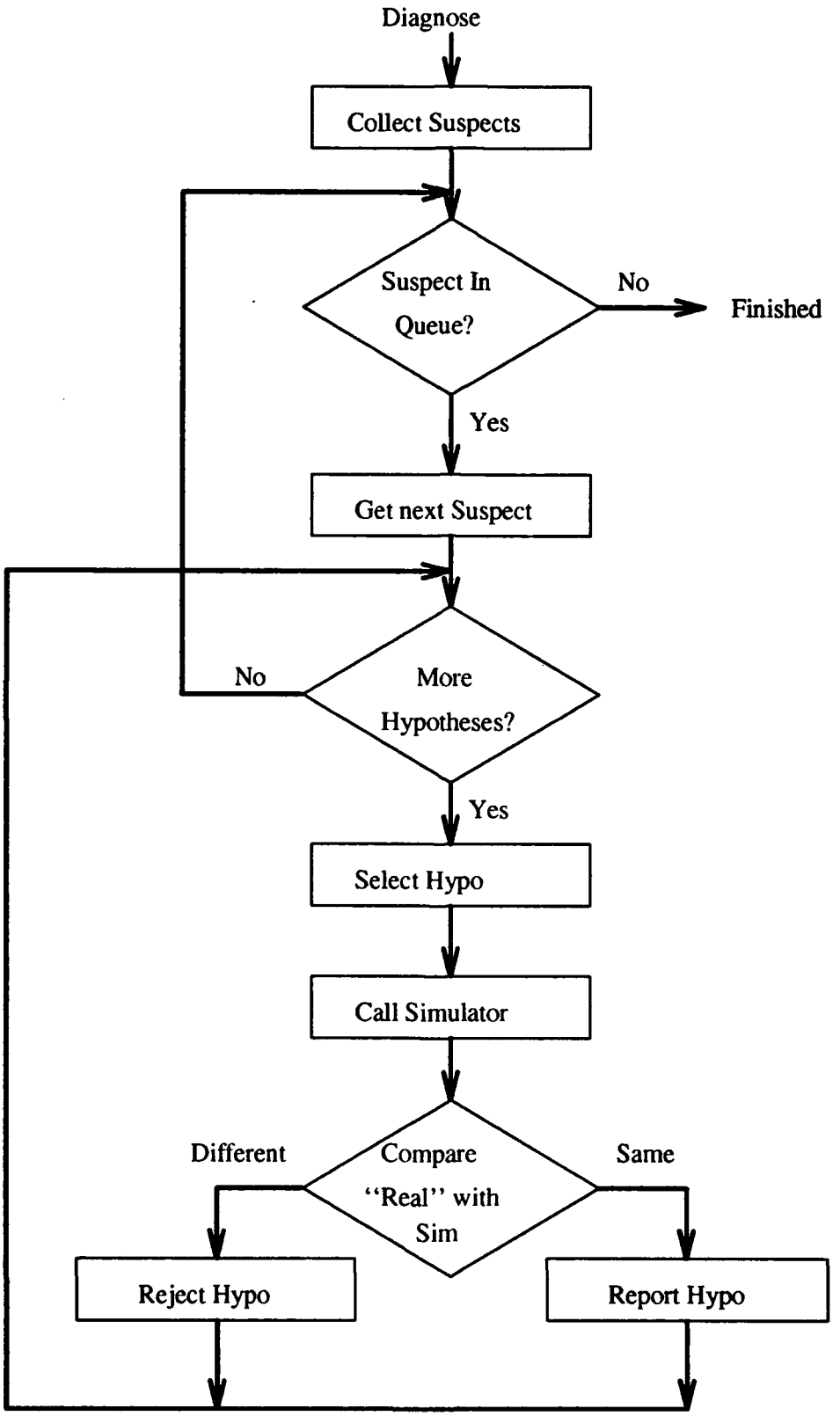


Figure 8. Calvin's Diagnostic Algorithm

```
- Entity declaration for 1/6 of 74xx04 inverter
entity 7404_inv is
    port(
        A: in bit;
        Y: out bit
    );
end 7404_inv;
```

Figure 9. Entity Declaration for 7404 Type Inverter

```
- Architecture body for 1/6 of 74L04 inverter
- Propagation delay determined by the average of pLH and pHL
- as given by the TTL Data Book, Vol. 2 by Texas Instruments
architecture 74L04_inv of 7404_inv is
begin
    process
        Y <= not(A) after 33 ns;
        wait on A;
    end process;
end 74L04_inv;
```

Figure 10. Architecture Body for 74L04 Inverter

that represent the parsed source code. The grammar of the VHDL subset used in this research is described in Appendix A.

The first part of the source code file contains *entity* and *architecture* declarations. The *entity* declaration defines the components that make up a circuit, along with their interfaces. The actual workings of the component are described by the *architecture* declaration. Since there can be many ways to describe the internal workings of a component, there can exist more than one *architecture* declaration for an *entity* declaration. For an example of multiple architectures for the same entity consider the TTL 7404 inverter. A possible entity declaration for the gate is shown as Figure 9. The characteristics of the gate vary depending on the technology used. An architecture body for the 74L04 inverter is shown in Figure 10. while the architecture body for a 74S04 inverter is described in Figure 11. The parser maintains a list of currently defined entity declarations and architecture bodies that have been declared in the source code file.

```
- Architecture body for 1/6 of 74S04 inverter
- Propagation delay determined by the average of μLH and pHL
- as given by the TTL Data Book, Vol. 2 by Texas Instruments
architecture 74S04_inv of 7404_inv is
begin
    process
        Y <= not(A) after 4 ns;
        wait on A;
    end process;
end 74S04_inv;
```

Figure 11. Architecture Body for 74S04 Inverter

```
configuration decode_11con of decode is
    for structural
        for I1: inv use configuration work.invcon;
        end for;
    end for;
end decode_11con;
```

Figure 12. Configuration for *Structural* of *Decode* (4:123)

The VHDL simulator requires a set of processes and their interconnections. The code for the processes is generated when the parser finds process definitions within the architecture bodies. After the code for a process has been generated, it is handed over to the diagnostic module. The diagnostic module generates possible fault hypotheses for the process, and returns it to the parser. The parser takes the correct behavior and the hypothesized faults, and collects them into a block for the simulator to use.

Which architecture is used by the simulation is determined by the *configuration* source code. An example configuration is shown as Figure 12. In this implementation, the parser expects a VHDL configuration to be at the end of the source code file. When the parser has reached the end of the source file, it returns control to the main program.

```
Initialize activation record queue
while queue not empty:
    Get next time
    while new time == time of next record on queue:
        Compare new value of signal described by the record with the
            current value. If they are equal, throw out record and loop.
            If not, add record to set of signals to update.
        Set value of signal to new value described by record
        Collect all behavior instances whose input is connected to
            this signal.
    For each behavior instance collected:
        Determine which process code is to be executed
        Execute code, posting new signal values to the queue
```

Figure 13. VHDL Simulator Pseudo-code

*3.2.2 VHDL Simulator* The simulator is based on an Intermetrics VHDL system. This VHDL system was described by Comeau in Chapter III of his thesis (3:41-61). The basic operation of the simulator is described by Figure 13.

The simulator revolves around a priority queue that contains information for updating the various signals in the simulation. These *activation records* contain information on which signal is to be updated next, and how it is to be updated.

The first time the simulator is run, a routine is called which places activation records for all the signals present in the simulation into the queue. Each is given a default value and an update time of 0. This simulates the circuit being "switched" on for the first time. For subsequent invocations of the simulator, the input signals to the system are placed in the queue. After the queue is initialized, execution continues into the main loop.

The VHDL main loop first updates the system clock to the value of the top record in the queue. Then all records that have this new time are collected. For each of these records the new signal value is compared to the current value. If the values are different, the simulator determines which behavior instances are connected to the signal. The signal value is then changed to its new

value. If they are equal, the behavior instances connected to this signal perceive no change, and do not need to be updated.

After all the records at the new time have been processed, the simulator updates the affected behavior instances. Since there may be many separate instances that all refer to same process code, the simulator must first determine which process to execute. The process contains instructions on how to simulate the process's behavior. During execution of the process, new values may be placed on output signals. The process execution module creates a new activation record for the signal and places it on the queue.

After all the affected behavior instances have been updated, the simulator loops until the queue is empty. This signifies that the circuit has reached a stable state. The output values can then be checked.

*3.2.3 Diagnostic Routines* The diagnostic routines are divided into four sections. While parsing the source code, a module generates fault hypotheses for the processes. Another module generates suspected bad components based on comparing the simulated outputs with the actual outputs. A third section determines which hypothesis to use for the suspect. Finally, there is a section to re-simulate the circuit and determine if the hypothesis for the suspect is valid.

This research effort used the Scarl's "Full Consistency" algorithm as modified by Dries. One goal of this effort was to make Calvin modular enough so that the various modules could be upgraded or replaced as needed. To do this, the diagnostic modules were written as loosely coupled modules. There is little or no parameter passing between them, other than the values of the input and output signals.

*3.2.3.1 Hypothesis Generation* Fault hypotheses are generated during the parsing of the VHDL code. After each VHDL process is created from the source code, the parser calls the fault hypothesis module. This takes the current process and generates possible faulty behaviors for

the process. The faults are predetermined by Calvin. Calvin generates faults for each output stuck high, each output stuck low, each input stuck high, and each input stuck low. Only one fault at a time is allowed in the circuit.

*3.2.3.2 Suspect Collection* Suspect components are those that can affect the reading of the sensor that differed from the simulated circuit. This limits the suspects to those that directly or indirectly drive the sensor. In Dries' and Scarl's algorithms, this is done by collecting each component upstream starting from the faulty sensor. In this research this is done with a simple depth-first search, starting with the faulty output.

*3.2.3.3 Fault Generation* After the possible suspects have been identified, it is the job of this module to "break" the test circuit. The *collect suspects* module creates a queue that contains all the possible suspects that could be the cause of the fault. The fault generating module takes the suspect at the front of the queue and causes it to break. It iterates through each fault hypothesis generated during the parse. After the re-simulation, the outputs of the simulated faulty circuit are again compared with those supplied from the "real world." If they match, the candidate hypothesis is kept. If not, the candidate is rejected.

*3.2.3.4 Re-simulation* After a possible fault hypothesis is selected, the circuit must be re-simulated. The fault generating module rearranges the pointers to the process code blocks, and re-initializes the simulator variables. This allows the same VHDL simulator that generated the results for correct operation also to be used for the simulation of the faulty circuit.

## *3.3 Implementation*

*3.3.1 Selection of a Programming Language* Since I had decided on an object-oriented approach, the programming language must be able to support object-oriented programming. The

language must be powerful enough to accomplish the task. I must also be able to use the language. Some criteria for the language are:

- **Object Oriented**

  As described above, several parts of Calvin are inherently object-oriented. An object-oriented approach also makes information hiding and modularity easier. Since the only way to access or change the hidden information is to use explicit calls to an access function, it is easier to find logic errors in the program. This also prevents inadvertent tampering with the information.

  Certain aspects of the object-oriented paradigm were not initially thought to be necessary. These included the concept of inheritance. It was later found that inheritance could be used in the parser, simplifying and standardizing the data structures greatly.

- **Ease of Prototyping**

  Since most of the code was to be generated from scratch, several false starts were anticipated. Code must be easily and quickly written, without having small changes requiring massive rewrites. Some data must be able to move through the system without having to worry about type-casts. This feature can have some disadvantages: this type of programming can lead to poorly-written code that may lead to recoding complications and hard-to-find bugs.

- **Convenient Development System**

  The language should have a complete set of development tools. Although this does not necessarily mean an integrated development environment, the basic tools for editing, compiling, running, and debugging should be present and work together smoothly. For accessibility, some or most of the code should be able to be developed on a MS-DOS based machine.

- **Compatibility with LEX/YACC**

  As discussed later, a grammar for the VHDL parser was obtained from the University of Cincinnati. This grammar was written in Bison, a GNU version of the UNIX utility YACC.

Bison is the GNU version of the UNIX utility YACC (Yet Another Compiler-Compiler). These programs take a language grammar and generate C code that parses that language. The C code is then compiled and linked with the rest of the program. The main advantage of Bison over YACC is that it allows larger language grammars to be parsed.

Bison and YACC work with another utility, LEX. LEX takes a description of the tokens recognized by a language, and generates C code that parses these tokens. The source files for both LEX and Bison contain embedded C code that is inserted into the output files. This code determines the actions that take place when certain keywords or structures have been parsed. By using Bison and LEX with the VHDL grammar from the University of Cincinnati, I did not have to write the actual parser. My task was limited to adding the actions to take place one certain VHDL constructs were recognized.

The system must be able to take the output code files from Bison and LEX and link them into the simulator/diagnostic routines. Alternatively, a separate program could be written that would parse the VHDL source and pass the resulting information to the rest of the system through a file. In any case, some code would have to be written using the C language (note that this turned out to be not quite true; A GNU version of LEX, FLEX, was modified to use C++).

- **Familiarity**

  Last, but definitely important, the programmer must be familiar with the language. Time spent learning a new language is time that could not be used on the research effort.

Initial candidates for the programming language were Ada, C/C++, LISP, and PC-Scheme. A discussion of these languages considering the above criteria follows:

- **Ada**

This language is the DOD standard language for new programming efforts (2:8). Although some claim it is an object-oriented language, there is some debate. It does not support inheritance, but as noted, inheritance was not thought to be important in this effort. It does have good support for modularity and information hiding, the desired object-oriented features most needed for this effort. A main feature of Ada is enforcing type-checking. Although this can lead to more reliable code, it can hamper prototyping efforts. The type-checking makes it possible to catch logic errors earlier in the development process. A more serious problem is its efficiency. Ada does run on MS-DOS platforms; however, the edit/compile/run cycles tend to be a lot longer then those of the other languages. Ada does have methods for linking in modules from other languages, so it should be compatible with LEX/YACC. Ada's syntax, being based on Pascal, is not greatly different from other common programming languages.

- **LISP**

  This language is very much associated with artificial intelligence research. In contrast to Ada, LISP has very little type-checking. It also provides a high level of abstraction not found in "lower-level" languages such as C. It tends to be more compact than equivalent programs in other languages. Features such as not having to declare variables until they are used for the first time allow very rapid programming. Although LISP is not object-oriented by itself, the *Flavors* extensions add this capability to LISP. A major problem with LISP is the size of the language; it does not fit well on a MS-DOS platform. Programming in LISP also requires a different mind-set than more traditional languages.

- **PC-Scheme**

  PC-Scheme is a variation of LISP that runs on MS-DOS machines. It has the same programming style as LISP, allowing rapid prototyping and smaller programs. It comes with a well-integrated environment for programming in PC-Scheme. It also has hooks for integrating outside code modules.

- C/C++

  C is sometimes referred to as a "low-level" programming language. It is a powerful, but dangerous language. Like machine language, C assumes the programmer knows what he is doing, even when he doesn't. Examples include lack of bounds checking on arrays, and little type checking on parameters. Unless care is taken, this can result in obscure bugs that can affect areas of the code far away from the original problem. C itself is not object-oriented, although object-oriented techniques can be used.

  C++ is a superset of the C language that adds several features, such as object-oriented structures. Although the programmer can still cause obscure bugs, C++ has several features that tend to catch problems earlier during development. These include function prototypes and type-safe linkage, which specify function parameter types and return values.

  C/C++ is available on a wide selection of platforms, including both the Sun Sparcstations and MS-DOS machines. The UNIX operating system includes many tools for using C-based projects. There are also good development systems for MS-DOS platforms, such as those by Borland and Microsoft. Another advantage is that the output of Bison and LEX are C source code files. By using C or C++ integration of the various parts of Calvin would be much simpler, without any concerns about cross-language interfaces. Finally, I have had much experience in C programming, as well as some with C++.

After evaluating the languages, the initial selection was to use C++ for the simulator and C for the parser. The simulator was the most object-oriented, and would be best written in a language that supported such constructs. Since the output of Bison and LEX were C files, it was thought that the supporting modules for the parser also should be written in C for ease of integration. The diagnostic functions were rather loosely coupled to the rest of Calvin. C++ was chosen for these functions to take advantage of the additional power and type-checking features of C++.

The initial configuration was a parser module separate from the rest of the system. This would be run on a UNIX system using GNU Bison and standard UNIX LEX. The rest of Calvin would be developed on a MS-DOS machine using the Borland C++ 3.0 development system. The Borland environment contained both an ANSI standard C compiler and version 2.1 of C++, which included templates (the equivalent of the Ada Generic structure)(15:33). It also came with libraries for container classes. Familiarity with Borland products also resulted in a shallow learning curve, allowing more time for development.

During development a copy of Bison was found for MS-DOS machines. After finding that is was functionally equivalent to the version running on the UNIX system, the Bison code was ported to the MS-DOS platform. A GNU version of LEX, called FLEX, was also found that would run on MS-DOS machines. At this time, all code was ported to the MS-DOS platform, with the parser being compiled in C and the rest of Calvin in C++. By modifying the skeleton file for Bison, the output source files could be compiled by the Borland C++ compiler. Now, all the source files could be compiled by one compiler into one program. By doing this, integrating the various modules became trivial. An additional benefit was allowing C++ functionality in the parser section.

*3.3.2 Implementation Details* The following sections describe how Calvin was implemented. As described previously, Calvin can be split into three main areas: the VHDL parser, the simulator, and the diagnostic routines. Calvin was designed so that the three areas are relatively independent of each other and can be easily expanded.

A full-adder circuit is used as an example throughout this section. Figure 14 shows a schematic for the full-adder. This example was taken from VHDL: Hardware Description and Design (12:18-22)

The source consists of descriptions for an OR gate, a half-adder, and the full-adder. The VHDL source code for the full-adder is contained in Appendix B.1.

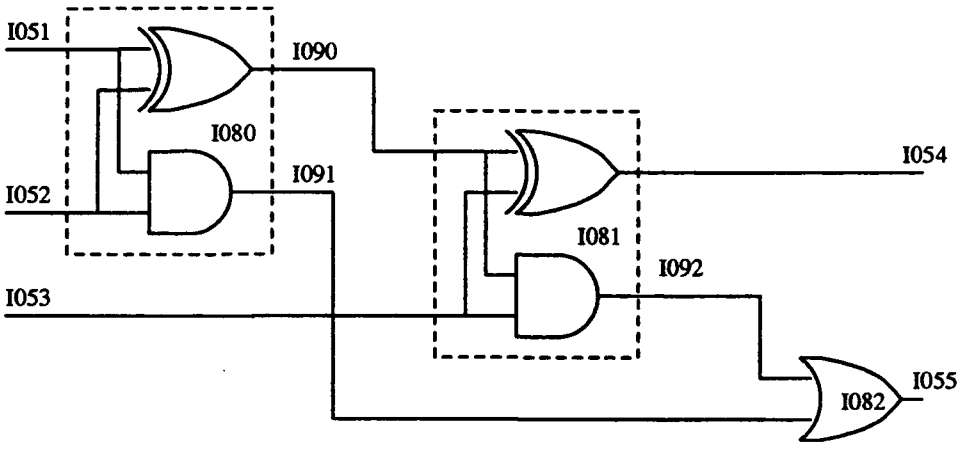


Figure 14. Full-adder Schematic

### *3.3.3 VHDL Parser*

*3.3.3.1 Introduction* The parser was built around a VHDL grammar written for the Bison compiler-compiler. As the VHDL source file was parsed, several data structures were built that represent the source code. This was done by embedding in the Bison grammar file calls to outside modules that build the data structure as the various VHDL constructs were parsed. At the end of the VHDL source file were configuration statements. These were handled by embedding calls in the Bison grammar to call routines to build up the simulator objects. When the end of file was reached, the parser surrendered control back to system.

*3.3.3.2 Bison Code* The University of Cincinnati VHDL grammar was for the IEEE-1076 specification of the VHDL language, with a few modifications. These were done so that Bison could generate a parser for the language. Bison generated a LALR(1) parser, which could not parse the entire VHDL language as described by the IEEE specification. These modifications are summarized in the header to the Bison code, which is in the file **UV**, in Appendix D.1. The original code had three shift/reduce conflicts and three reduce/reduce conflicts. Since signal types in Calvin were limited to type BIT, the number of shift/reduce conflicts was reduced to two and

reduce/ reduce conflicts to one. These modifications are documented in the part of the grammar that parses the VHDL **Type** token.

As the VHDL source was parsed, an internal representation was built. As each VHDL construct was recognized, the relevant information was stored in that representation. There was a module that maintained the current information for each VHDL construct. This was kept in dynamic memory. When the parser recognized the start of the construct, the current information was set to a default state. As the construct was parsed, the parser called functions that added the newly acquired information to the current construct. After the parsing of the construct was finished, a pointer to the finished construct was passed back, usually to a field within a higher-order construct. Some constructs, such as signal and port lists, were passed as linked-lists.

Figure 15 shows an example entity description for an OR-gate. After the parser found the **is** keyword, the grammar dictated that a port clause would follow. A call was made to port_clear() to initialize the current *port* data structure. The keyword **port** and the "(" token were then recognized. The parser then looked for a formal port list. This consisted of an identifier list "i11", colon token, a direction (**in** or **out**), and a signal type (**bit**). These values were placed in the appropriate fields within the *port* data structure. The parser then looked for the ")" and ";" tokens. At this point the *port* data structure was complete. A pointer within the module tasked with constructing this structure pointed to the memory block that contained the information. The parser then called a function that took this pointer and placed it within the current *entity* data structure. This continued until all the VHDL source had been parsed.

*3.3.3.3 FLEX Code* The parser generated by Bison required a module to recognize the tokens and keywords in the source file. This module was generated by a lexical analyzer, FLEX. FLEX was a GNU version of the standard LEX program present in most UNIX systems. For the purposes of this research the two were equivalent. The chief advantage of FLEX was that there was a version that runs on MS-DOS machines. A few modifications had to be made so that

```
entity i15 is .
port(
     i11: in bit;
     i12: in bit;
     i13: out bit
);
end;
```

Figure 15. OR-gate Entity Description

| | |
|---|---|
| Identifiers | Identifier must be the letter I followed by a three digit number (I002, I234) |
| Integers | Sequence of digits |
| Reals | Not permitted |

Figure 16. FLEX VHDL Limitations

Borland C++ could compile the output file from FLEX. These modifications are summarized in Appendix C.

The input file to FLEX described how the tokens of the grammar were to be recognized. Also included were the keywords and tokens for the VHDL grammar. The tokens and keywords style for VHDL and Ada were similar. This made it possible to take a LEX file used in the CSCE663 *Compiler Theory and Implementation* course and modify it. This file was jointly written by Captain Chester A. Wright and me. The additional keywords required by VHDL were added to this file. The file is named **UV.LEX**, and is in Appendix D.2.

To speed development, several features of VHDL were restricted. The chief of these were identifier names. Identifiers throughout Calvin were defined as integers. To make it easier to come up with the handles, the VHDL identifiers were defined to be the letter 'I' followed by a three digit number. The features restricted by the FLEX input file are summarized in Figure 16.

A complete description of the supported VHDL grammar is given in Appendix A.

*3.3.3.4 Internal Data Structures* As each VHDL construct was parsed, the necessary information was recorded in the data fields of a corresponding internal data structure. As parsing

continued, the data structure might then be inserted into a field of a higher echelon structure. This structure could in turn become a field in an even higher structure. At the top of the hierarchy were two data structures: a list of entity declarations, and a list of architecture declarations. Figure 17 shows the complete hierarchy of the entity declaration, while 18 does the same for the architecture structure. The individual modules that make up these hierarchies are in Appendix E. The names of the modules correspond to the objects in the hierarchy.

Since this section was so tightly bound to the Bison module for the parser, it was originally written in straight C code using an object oriented style. At the time it was originally written, the GNU Bison was being used to generate C code. It was later ported to a MS-DOS system and compiled with the same C++ compiler as the rest of Calvin. Unfortunately time did not allow this module to be rewritten in straight C++ code. This would have resulted in more consistent data structures, resulting in more robust code that is easier to modify (and debug!).

Since an object-oriented approach was used for the data structures and associated functions, each of them tended to have the same structure. The routines for each structure were collected into a single module, separate from any other structure. The actual data structure itself was stored in the heap memory. Each module contained an internal pointer that pointed to a current instance of the structure. This structure, known as the *current* structure, was the one that was currently being parsed.

Each structure had a *clear* routine: this initialized the internal variables and set the internal structure pointer to **NULL**. For list-type structures there was a routine to create a new object and add it to the front of the internal list. Next, there were a set of functions that added the values to the structure. These functions were called by the parser as the appropriate value was determined. Sometimes these functions added pointers to other data structures; other times a value was added to the field. After the object had been parsed, there must be some way for the outside program to use the structure. Another function handled this task by returning the address of the current data

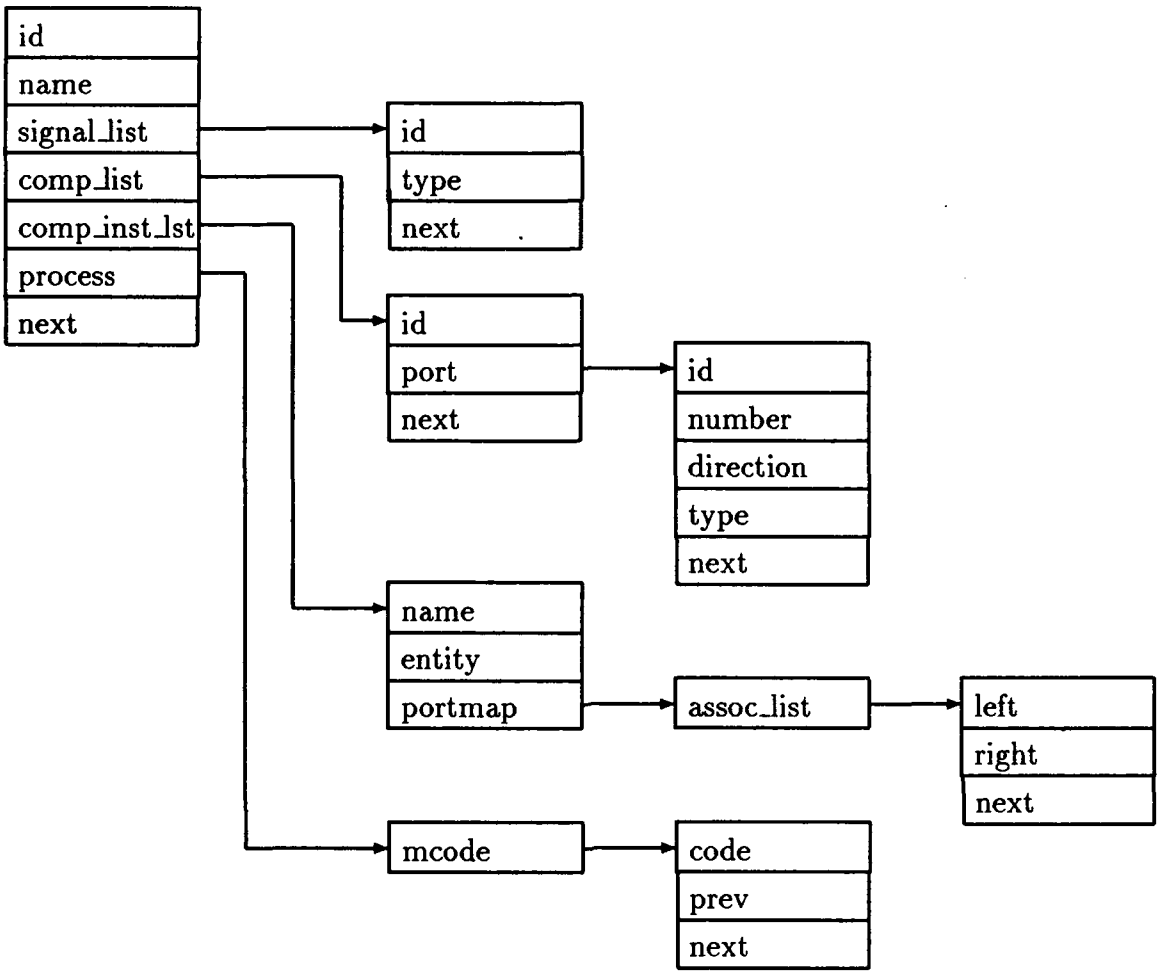


Figure 17. **Entity** Hierarchy

structure. Usually this was inserted into a data field of another data structure by corresponding functions in the higher structure. Finally, there were print functions that printed out the values of a data structure. One function displayed the current structure, while another described the one passed by a pointer reference.

*3.3.3.5 Translation to Simulator Data Structures* The parser translated the internal representation of the source code into class instances usable by the simulator when it parsed the VHDL configuration statements. In this effort, the configuration source must come at the end of the source file, after all the architecture and entity declarations have been parsed. As the configuration

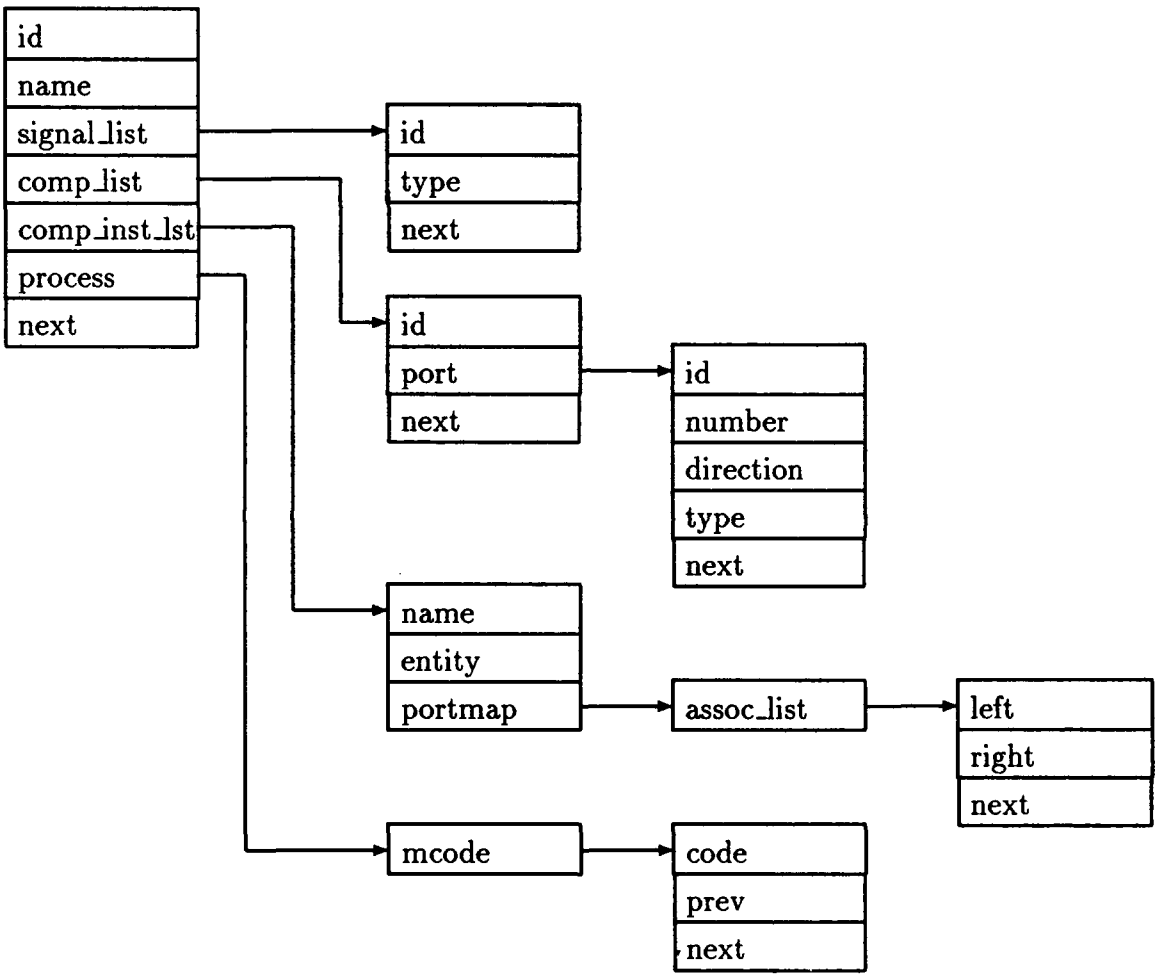


Figure 18. **Architecture** Hierarchy

information was parsed, Calvin generated the simulator objects. The information given in the configuration section determined the construction of these objects.

### *3.3.4 VHDL Simulator*

*3.3.4.1 Introduction* This section describes the VHDL simulator section of the Calvin. The flow of the simulator is first described, followed by a discussion of each of the main C++ classes used to implement the simulator.

*3.3.4.2 Overall Flow* The structure of the simulator is shown in Figure 19. These functions are in the module **VHDL.CPP**, in sections F.21 and F.22 in Appendix F. The simulator is built around a priority queue of activation records. Each activation record has three fields: the name of the signal to change, the new value of the signal, and when it is to be updated. Priority in the queue is based on the time stamp of the activation records, with earlier times towards the front.

The **Process_low_time()** function first sets the current simulation time by examining the top activation record on the queue. All activation records with the new time are then pulled from the queue. As each record is removed, the value of the signal specified in the record is checked against its current value. If there is no change, the activation record is ignored. If there is a change, the signal's value is set to the value in the activation record.

Signals are connected to objects that represent how an architecture behaves. In Calvin, these are called *Behave* objects. When a signal changes value, all Behave objects connected to that signal must be updated using **update_behave()**. These Behave objects are specified by the signal's **conns** list. The label **conns** is the equivalent to the "conns" that Comeau describes for the Intermetrics VHDL system. In the Intermetrics system, "conns" is the pointer to a list of behavior instances for which this signal is an input (3:50). The identifiers of these objects are collected into a **set** object. This is a container class defined in the Borland library that allows only one copy of

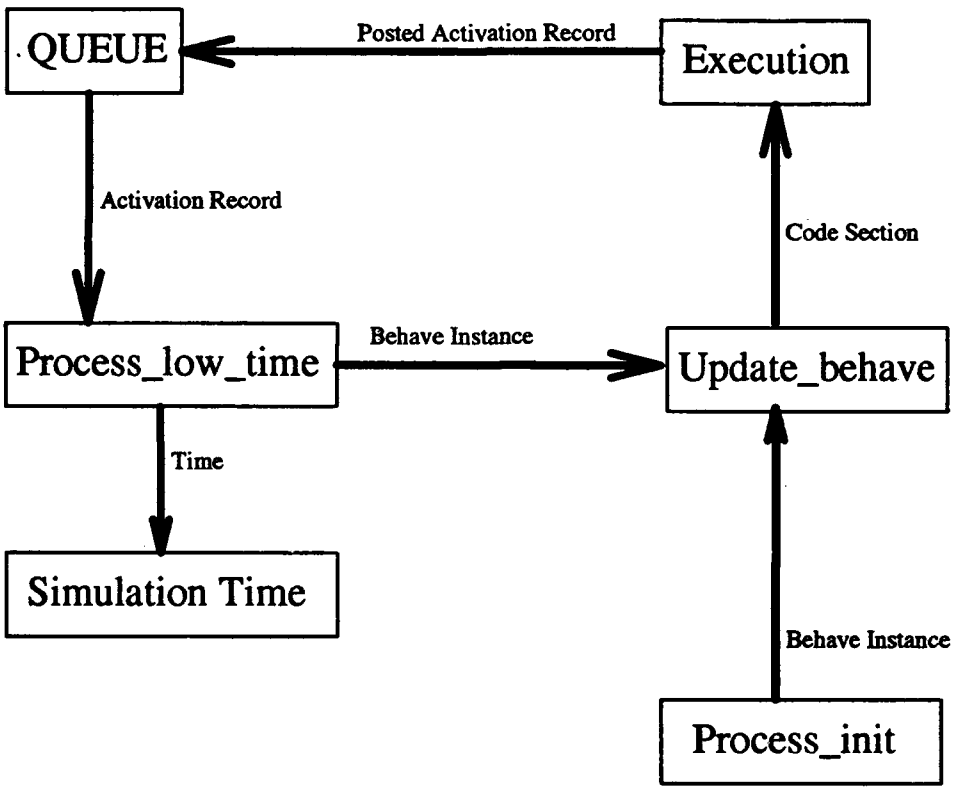


Figure 19. Block Diagram of VHDL Simulator

a member to be in the set. This keeps the simulator from updating a **Behave** object twice during the same simulation time.

After all the signals with the same simulation time have been removed from the queue, the simulator updates the Behave objects that the changed signals drive. Each *Behave* object references a **Block** object. This object contains a set of code sections that describe how the Behave object works. Then, according to the hypothesis generated during the parsing, the simulator takes one of the code sections from the block and executes it.

The code section is a list of opcodes, which are discussed in more detail below. The object can pull in current signal values through the input port list of the Behave object. Other opcodes perform calculations. The simulator handles output signal values by posting them to the priority queue. The **M_POST** opcode specifies the modified signal's name, its new value, and the propagation delay of the block. The simulator builds an activation record from this information and inserts it into the priority queue, completing the simulation cycle.

Since at the beginning of simulation time the queue is empty, there must be some way to start the simulator. The **process_init()** function handles this by calling **update_behave()** for each Behave object in the simulator. Any **Behave** objects that change a signal value will place an activation record in the queue. The earliest activation record is then pulled from the queue, starting the simulation. The simulator will then cycle until the queue is empty. This signifies that the circuit has reached a stable state. If the circuit has feedback, it is possible to design a circuit that will never be stable. This effort assumes only feed-forward combinatorial circuits; these will always reach a stable state.

*3.3.4.3 Microcode* The simulation of the circuit's components is handled by a "microcode" interpreter. This code is in modules **MCODE** and **CODE**, in Appendix F. Instead of generating C source code like the Intermetrics VHDL system, Calvin generates opcodes for an interpreted language, which will be referred to in this thesis as "microcode" or "mcodes.'

The microcode interpreter is stack-oriented with a separate set of registers. This style was inspired from the Hewlett-Packard line of programmable calculators, which I have used for several years. Operands are pushed onto an internal stack, which is then used by the operators. An auxiliary set of registers can hold values that need to be saved from change. A more complete description of the opcodes is discussed later.

*3.3.4.4 Data Structures* To speed development of Calvin a "few shortcuts" were taken. Several places within the various objects required a set of values. An attempt at using the **Set** class supplied by Borland was unsuccessful, so an array was used instead. In most cases, the object used an integer array along with an index variable that marked the next empty slot in the array. The length of the array was defined in the header files as **MAX_xxx_LEN**. Since values were only added to the array, no special garbage collection routines were needed.

As Calvin developed, the maximum array lengths were adjusted as needed. This method of managing data collections was wasteful of memory; however, lack of memory never was a problem.

Another shortcut dealt with identifiers. Using proper software engineering practices, there should be a separate identifier class. To speed the development of Calvin all identifiers were defined as integers. This also simplified the internal operand stack for simulating the process code sections. By having both identifiers and signal values defined as integers, the stack could be implemented as a simple integer stack. An alternative would have been to create a new data object that would be a union of identifier type and signal value that included a field determining which was which. Implementing future data types other than **BIT** will require this approach.

*3.3.4.5 Signal Record Class* The **signal** object is used to connect the various processes of the simulated circuit together. The module that defines the **signal** object is module **SIGNAL**, in Appendix F. The signal object in the simulator is derived from the Intermetrics' signal record as described by Comeau (3:3.10). Figure 20 describes the data fields for the signal record object. The

| | |
|---|---|
| id | Signal Identifier |
| name[ ] | String name for signal |
| cval | Value of signal |
| conns[ ] | Behave instances connected to this signal |
| last_conn | Last conns |
| driver_bi | Behave instance that drives this signal |

Figure 20. Data Fields for **SignalRecord** Object

| | |
|---|---|
| SignalRecord | Constructor |
| get_id | Return signal identifier |
| print | Print signal record (debugging function) |
| add_conns | Add behave instance to conns list |
| get_conns | Return pointers to conns list |
| set_cval | Set signal value |
| get_cval | Get current signal value |
| set_driver_bi | Set driving behave instance |
| get_driver_bi | Get driving behave instance |

Figure 21. Functions for **SignalRecord** Object

**ID** field uniquely identifies the signal in the circuit. The **name** field is used for the user interface. The current value of the signal is maintained by the **cval** field. The list of behave instances that are driven by the signal is kept in the **conns** array. These instances are those that will be updated whenever the value in **cval** is changed by a record being de-queued from the activation record queue. **last_conn** is an index into the **conns** array. It points to the next available slot in that array. Finally, **driver_bi** is the identifier of the behave instance that drives the signal.

A list of functions available for the SignalRecord object is in Figure 21. Like most objects, the ID of the signal can be obtained by calling **get_id()**. The functions **add_conns()** and **get_conns()** allow access to the list of Behave objects connected to the signal. The value of the signal is set by **set_cval()**, and obtained by **get_cval()**. Access to the name of the Behave object that drives the signal is through **set_driver_bi()** and **get_driver_bi()**. For debugging, **print()** was written to display the data within the signal object.

*3.3.4.6 Behavior Instance Class* When the VHDL simulator changes the value of a signal, that signal returns a set of circuit components that must be updated. Each of these com-

| | |
|---|---|
| id | Behave identifier |
| block_id | Block associated with this behave |
| code_select | Current hypothesis in use for simulation |
| input[ ] | List of signals tied to behave inputs |
| last_in | Last input added |
| output[ ] | List of signals ties to behave outputs |
| last_out | Last output added |

Figure 22. Data Fields for **Behave** Object

ponents is simulated by a **Behave** object. The source code for these objects is in the module **BEHAVE**, in Appendix F.

The private data fields in the **Behave** object are shown in Figure 22. As in most objects, the Behave object contains an identifier **id**. Each Behave object also contains a set of input signals and a set of output signals. In Figure 22 these are the fields **input[ ]**, **last_in** and **output[ ]**, **last_out**.

The code for simulating the Behave object is contained within a separate **Block** object. The **block** object is defined in the **BLOCK** module in Appendix F. The ID of the **Block** object is kept in block_id field. Code_select determines which code selection to simulate in the **Block** object. This field is kept within the **Behave** object since there might be many instances referring to the same **Block** object, each selecting its own code.

The purpose of having both **Block** and **Behave** objects was efficiency. In VHDL there can be several instantiations of the same object. An example is an adder constructed with two XOR gates. When the XOR gate is defined in the source, a **Block** object is created to allow an XOR gate to be simulated. To avoid duplicating the **Block** for both XOR gates in the full adder, two **Behave** objects are created instead. Each of the **Behave** objects points to the XOR block with the **Behave** object's **block_id** field. Although the code section for XOR might not be long enough to justify breaking it out of the **Behave** object, other objects might be. This is especially true once the various fault hypotheses are included.

A list of routines that can be used on *Behave* objects is shown as Figure 23. There are two constructor functions. Both reset the indexes for the input signal and output signal arrays.

| Behave | Constructor |
|---|---|
| get_id | Return Behave identifier |
| set_code_select | Select code for execution |
| get_current_select | Get current code select |
| set_block_id | Set block identifier |
| get_block_id | Get block identifier |
| get_code_count | Get number of hypothesis |
| add_input | Add signal to input of Behave |
| get_input | Get ID of signal tied to input |
| get_input_count | Get number of input signals |
| add_output | Add signal to output of Behave |
| get_output | Get ID of signal tied to output |
| get_output_count | Get number of output signals |
| print | Print Behave (debugging function) |

Figure 23. Functions for **Behave** Object

In addition, one constructor allows the ID of the block to be set. The other constructor lacks parameters; this is required by C++ for creating an array of these objects.

The **Behave** objects include several access functions. The function **get_id()** returns the ID of the object. To hypothesize faults, or to use the correct, behavior requires a call to **set_code_select()**. Passing a value of 0 to the object through this function allows the component to be simulated correctly (no faults). The current fault number is obtained by sending the block object **get_current_select**. To determine which **Block** object is to be executed for the **Behave** object, the simulator calls **get_block_id()**.

A group of three functions handles access to the input signals for the object. The parser calls **add_input()** to add a new signal to the **Behave** object. To get the ID of an input signal, the simulator calls **get_input**. Finally, the current number of input signals is obtained via **get_input_count()**. A similar set of functions handles the output signals. Finally, for debugging purposes, a print function prints a description of the **Behave** object.

*3.3.4.7 Block class* The purpose of the **Block** class of objects is to hold references to the various code sections that could be run to simulate a particular VHDL architecture. The source code for this object is in module **BLOCK**, in Appendix F. These are kept as an array within the

| | |
|---|---|
| id | Behave identifier |
| sim_code_id[] | list of code ID's for this block (process) |
| last_code_no | Last code number |

Figure 24. Data Fields for **Block** Object

| | |
|---|---|
| block | Constructor |
| get_id | Get block identifier |
| add_code | Add new process code ID to code list |
| get_code | Get code ID from code list |
| get_code_count | Get number of hypotheses in this block |

Figure 25. Functions for **Block** Object

**Block** object. In all cases the code section that simulates correct operation of the architecture is in position 0. Figure 24 contains a complete list of the private data fields within the **Block** object.

The functions available to the **Block** object include a constructor, a function for returning the **Block**'s ID, and functions for adding and retrieving the code section ID's. A full list of functions is listed in Figure 25.

*3.3.4.8 Code Class* Each **Block** object contains at least one code object for simulating the operation of the architecture. The **code** object is defined in the **CODE** module, in Appendix F. The code object can be thought of as a "program" for simulating a process. It contains an ID to allow it to be referenced by the appropriate **Block**. The "program" is stored as an array of **MCode** objects, which are described later. Figure 26 contains a list of the data fields within the **Code** object.

The functions available for the **Code** object, which are summarized in Figure 27, are straightforward. The **get_id()** function returns the **Code**'s ID. The parser uses **add_mcode()** while creating the process block. The code is simulated by calling the **execute()** function. A debugging function **print()** lists the program to the screen.

| | |
|---|---|
| id | Code identifier |
| code_blk[] | Program storage |
| last_code_no | Next available line in program storage |

Figure 26. Data Fields for **Code** Object

| | |
|---|---|
| get_id | Get identifier for code |
| add_mcode | Add new mcode to program |
| execute | Execute program |
| print | Print program (debugging function) |

Figure 27. Functions for **Code** Object

| | |
|---|---|
| M_NULL | Null opcode |
| M_GET | Get signal (signal no.) |
| M_POST | Post signal (signal no., value, delay time) |
| M_PUSH | Push |
| M_NOT | NOT (value) |
| M_AND | AND (value1, value2) |
| M_OR | OR (value1, value2) |
| M_XOR | XOR (value1, value2) |
| M_END | End execution |
| M_NAND | NAND (value1, value2) |
| M_NOR | NOR (value1, value2) |
| M_POP | Pop (and discard) value on top of stack |
| M_STORE | Store (addr) value into register |
| M_RETRV | Retrieve (addr) value from register |

Figure 28. **MCode** Op Codes

*3.3.4.9 MCode class* The purpose of this class is to gather all the available operations together. The only data field is the opcode. The opcodes are defined in the header file for the class. A list of them is shown in Figure 28. The complete class definition is in the **MCODE** module, listed in Appendix F. The opcodes are negative values. If the simulator encounters a positive value, it is interpreted as data and pushed onto the stack. When the code section for the process is finished, the **M_END** opcode signals the simulator to stop.

**M_NULL** is the equivalent of a **NOP**. It was present for debugging the simulator. **M_GET** and **M_POST** handle passing signal values into and out of the process. In both cases the signal number used by the instruction is on the top of the stack This number is an index into the input or output signal arrays in the executing **Behave** object. **M_POST** requires two additional parameters: the new value for the posted signal, and how far in future will the new value be assigned to the signal.

| MCode | Constructor |
|---|---|
| execute | Execute the instruction |
| print | print mnemonic (Debugging function) |
| get_op_code | Return op code |

Figure 29. Functions for **Code** Object

**M_POST** creates an activation record using the signal name, value and delay time. This delay time is added to the current simulator time in order to determine were in the priority queue the new activation record will be placed.

**M_AND**, **M_OR**, **M_XOR**, **M_NAND**, and **M_NOR** perform their named operations using the top two operands on the stack. The result is placed back on top of the stack. **M_NOT** inverts the value of the top of stack. A '1' value is changed to '0,' while a '0' is changed to a '1.'

Some hypotheses require an opcode for ignoring the current value of a signal. The **M_POP** opcode handles this by discarding the current top of stack. The new value can then be pushed onto the stack.

For certain faults, a value below the top of the stack may need to be changed to reflect a certain fault. The top of the stack can be saved and later restored by using the **M_STORE** and **M_RETRV** opcodes. These codes save the top of stack in a specified register. The top of stack can then be removed, and the value below changed. These opcodes also can be later used to implement temporary storage for other uses not required at this time.

Figure 29 contains the list of functions for the **MCode** class. The **execute()** function executes the instruction. The value of the opcode is returned via **get_op_code()**. The function **print()** was used to debug the system.

*3.3.5 Diagnostic Routine* This routine implements a version of Dries' Diagnose algorithm. A flow diagram of Calvin's implementation is in Figure 30. (This is the same diagram as Figure 8.) The algorithm is contained in the module **CALVIN**, listed in Appendix F.

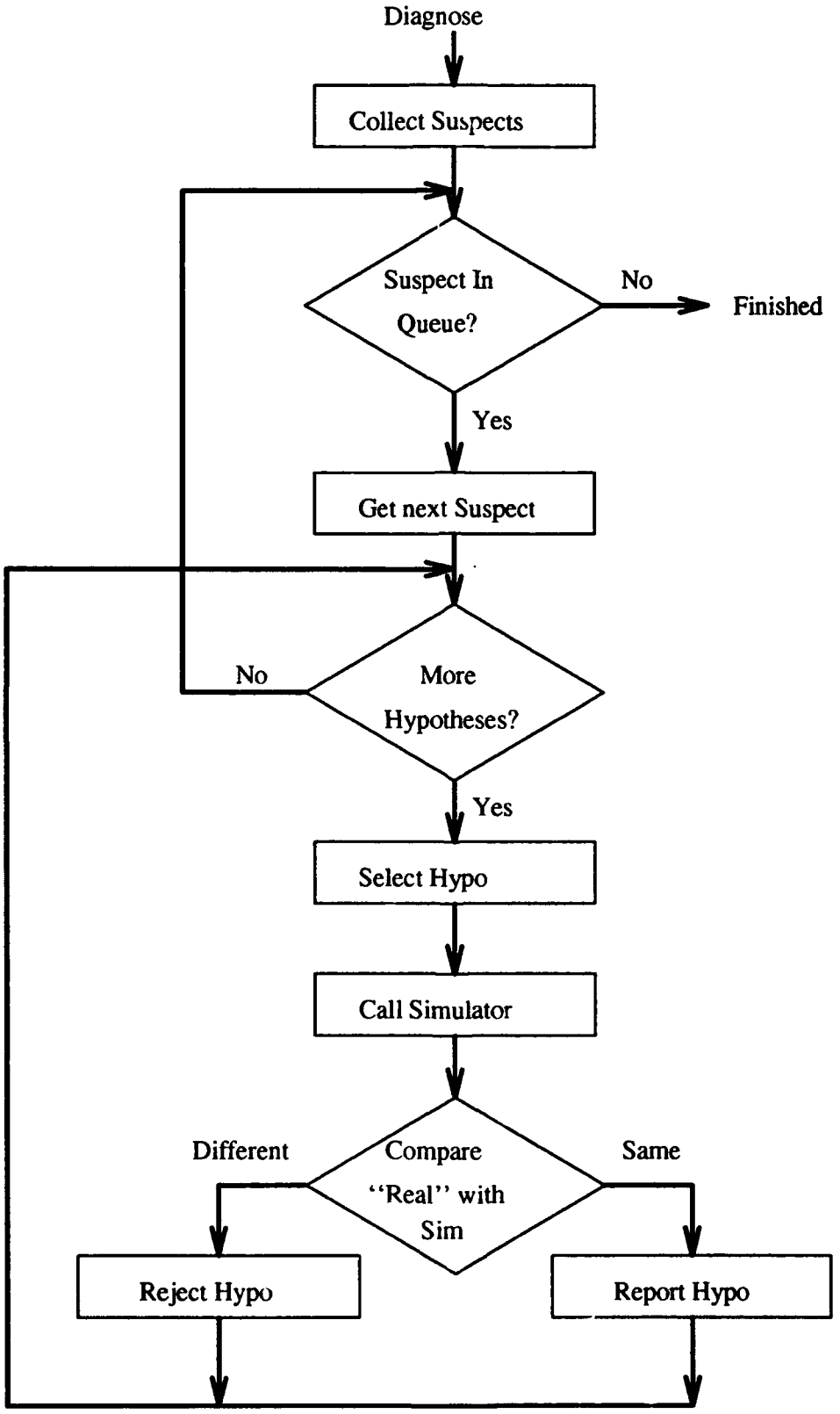


Figure 30. Calvin's Diagnostic Algorithm

*3.3.5.1 Fault Determination* During the diagnostic phase Calvin is given a list of input values and recorded output values from the "outside world." Calvin sets the circuit's inputs to those supplied and simulates the circuit. Calvin checks each sensor to see if it matches the recorded value. If all match, Calvin declares that the circuit has no apparent problems.

*3.3.5.2 Collection of Suspects* If an output value does not match the expected value, Calvin calls a routine to collect possible suspects. This routine takes the parsed representation of the circuit and determines which component could affect the errant output. These are placed in a queue.

Currently, suspects are generated by using a depth-first strategy. The component connected to the output is placed first in the queue. Then the collection routine is called recursively for each signal attached to that component's inputs. Recursion ends when it reaches an input signal. Although not very efficient, this module is almost totally independent of the rest of Calvin, and can be easily modified or replaced. The only output is the queue containing the list of suspects. It can be rewritten without affecting the rest of the system.

*3.3.5.3 Disproving Hypotheses* Calvin takes each suspect from the queue and modifies it according to the hypotheses generated during parsing. Each suspect body has a list of behaviors that were given it while it was being parsed. The fault is simulated by changing the active behavior to one of the fault behaviors. Calvin iterates through these hypotheses, re-simulating the circuit after the behavior has been switched. Calvin then checks all the simulated outputs against those that were supplied. If they match, Calvin prints a message stating the component's name and the hypothesis' name. Calvin then continues with the rest of the hypotheses. When all the hypotheses are finished for a suspect component, the behavior of that component is reset to the correct behavior. Calvin pulls the next suspect from the queue and the cycle repeats.

## *3.4 Summary*

This chapter gave an overview of Calvin. This included the generate and test areas. Next was a more detailed look at Calvin, breaking it down into parsing, simulating and diagnostic routines. Following this was a discussion of the implementation of Calvin. This included reasons behind the languages selected, and detailed descriptions of the parser, simulator, microcode, data structures and diagnostic routines. The source code for Calvin is in Appendices D through F. The next chapter will describe some test files used, and will discuss ways that Calvin can be improved.

# *IV. Results*

## *4.1 Testing Calvin*

Three circuits were used to test the diagnostic powers of Calvin. These were a full adder, a two-operation ALU, and an adder.

Figure 31 is the schematic for the full-adder. The schematic for the ALU is Figure 32 and for the adder is Figure 33. In Figure 32 the dashed lines are the probes inserted into the circuit. The actual VHDL code for the full-adder is in Appendix B.1, the ALU without probes in Appendix B.2, the ALU with probes in Appendix B.3, and the adder in Appendix B.4. Appendix B.5 describes a five-bit two's compliment ALU that performs addition and subtraction. This circuit is not discussed in this chapter; it was included as another example that Calvin can use. To validate these files, they were processed with the Zycad VHDL system. Since Calvin does not have libraries implemented, minor modifications were made to the files. Complete details of the modifications are in Appendix G. Appendix G also contains the results from the Zycad system. The example circuits performed as expected.

In Calvin all VHDL identifiers are limited to the character 'I' followed by a three-digit number. To keep track of the various components, the names were kept consistent.

The single-bit full adder was used during development. Although small, this circuit contained all the elements supported by the VHDL simulator in Calvin.

A two-operation ALU simulated a system with multiple independent subsystems that had little interaction. The ALU performs either an AND or an OR function on the two sets of inputs, depending on a select line. Since these are logical functions, the operation on one bit does not affect any other bit. The only area that affects all operations is the select line. The circuit was modified by routing the internal lines to outputs (sensors).

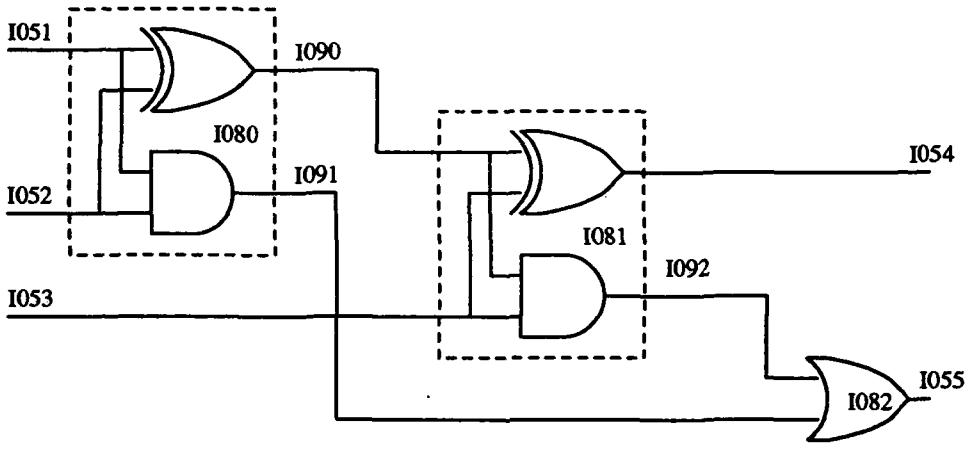


Figure 31. Single Bit Full-Adder Schematic

The four bit adder simulated a system in which the subcomponents interacted with each other. The adder consisted of multiple copies of the original full-adder, with the carry-out of one bit connected to the carry-in of the next. A faulty device will tend to affect many sensors.

*4.1.1 Running Calvin* This section discusses how Calvin was run. First is a description of how to run Calvin. Then two examples are given, the full-adder and the ALU. In the examples the correct operation of the circuit is validated. This is done by using sets of data that show correct operation of the "real" circuit.

Calvin took the source code for the test circuits and an input file that contained test inputs and outputs and attempted to find the problem (if one existed). The test outputs for the "actual" circuit were calculated before running Calvin. To validate correct operation of Calvin, this was done for correct operation of the circuit first. Then, Calvin was tested with errors placed in the circuit. The results of the errors were calculated and supplied to Calvin as the outputs of the "actual" circuit. In all cases Calvin did find the fault when the fault affected a sensor reading. Unfortunately, most of the time Calvin also would find many other possible problems that also could cause the same sensor values. Adding sensors to internal signals of the circuit cut down the number suspects. In effect this was adding probes to the test circuit.

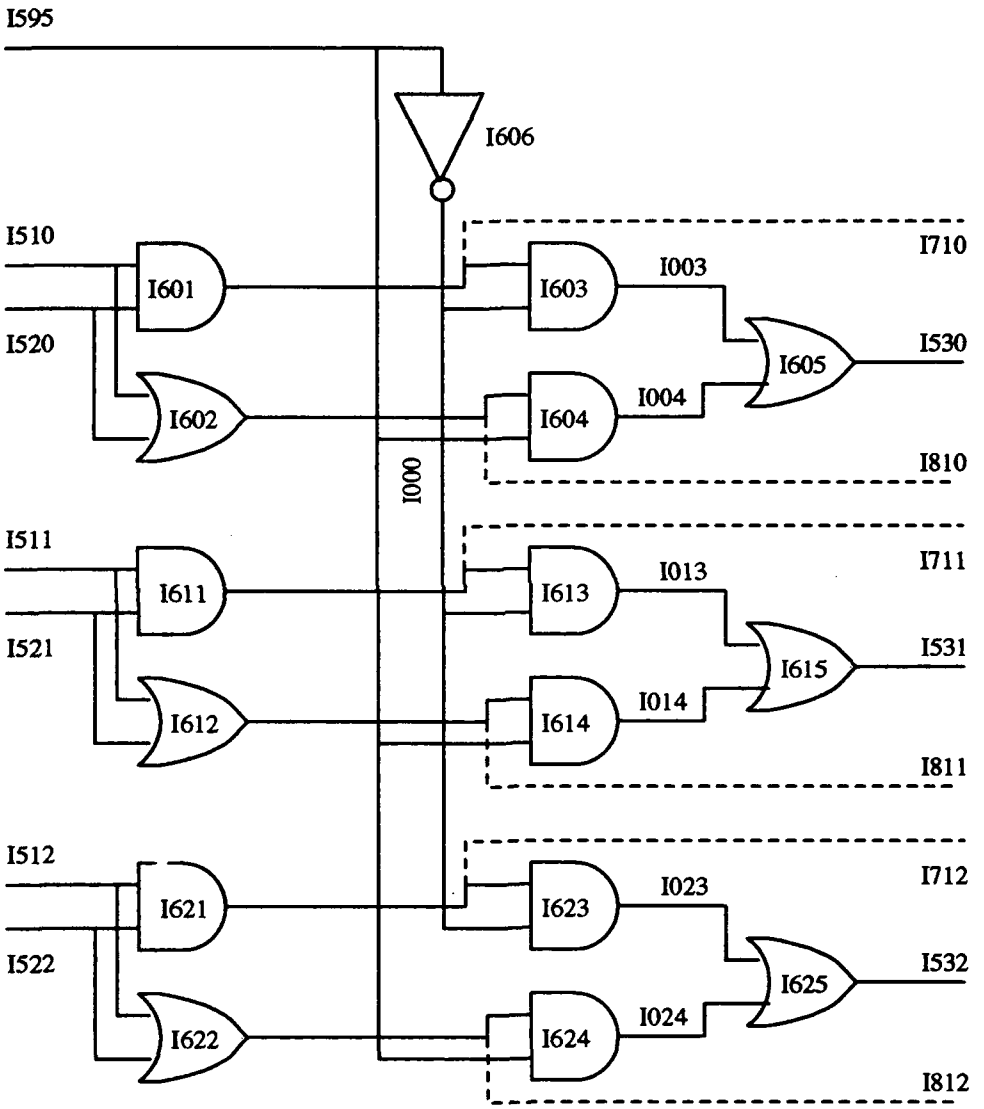


Figure 32. ALU Schematic

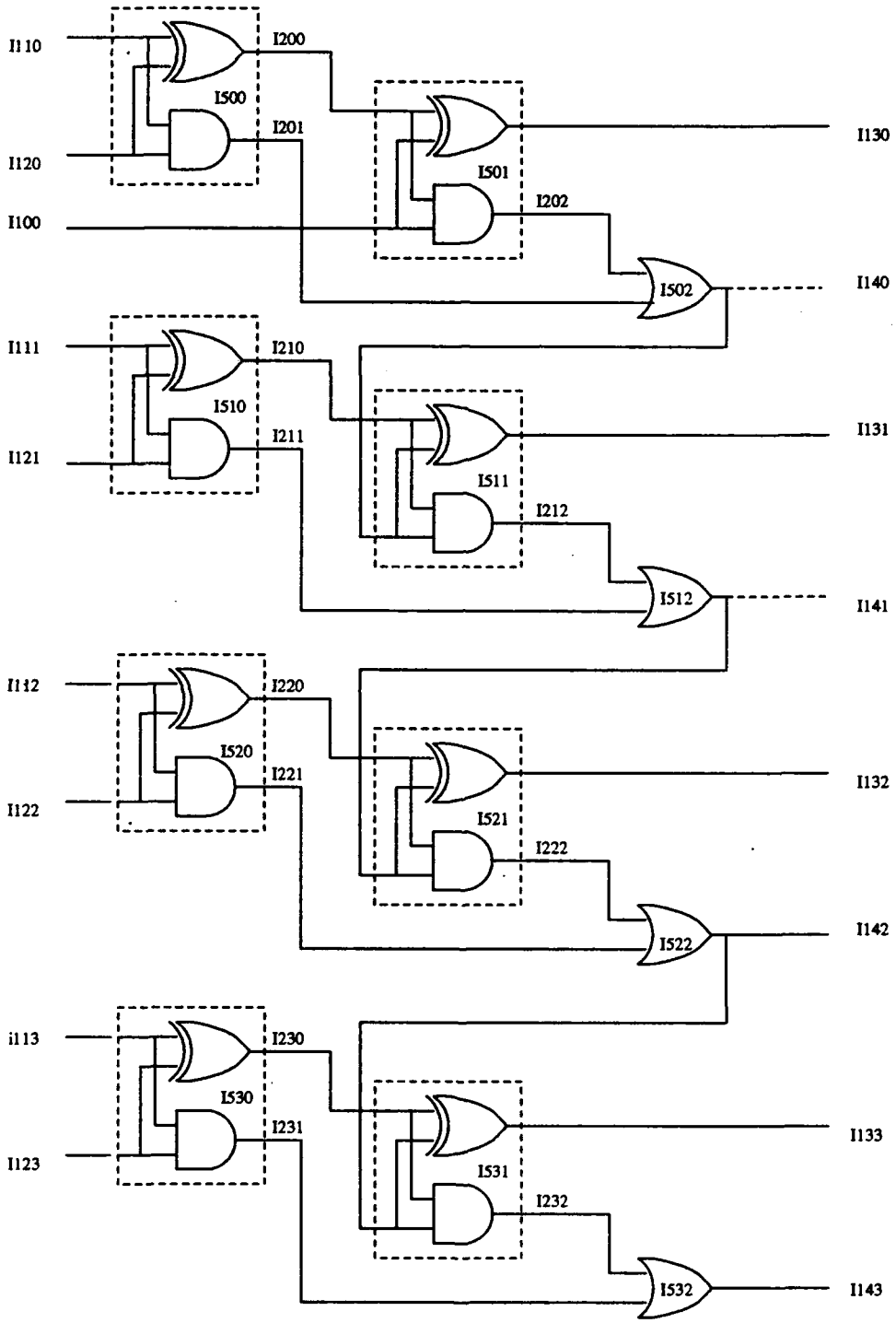


Figure 33. Four Bit Adder Schematic

The user interface to Calvin was designed so that in can be run in batch mode. The user supplies a set of flags (detailed in the module main.cpp), a source file, and a test file. The output could then be re-directed into a file and examined. This allows many test cases to be run at the same time. Most of the flags determined what information is displayed. The others controlled small improvements to Calvin. When Calvin determined that a suspect could account for all the outputs, it displayed it to the standard output device (screen or redirected-output file). After each diagnosis, Calvin printed out information about that run. This information included:

- how many suspect components that were collected,
- how many hypotheses that Calvin generated,
- how many different faults that could cause the supplied circuit's outputs,
- the number of activation records posted by Calvin's VHDL simulator,
- the number of **Behave** objects that were updated,
- total number of simulations done by Calvin.

Idealy, these numbers should be as low as possible. The first three numbers determine how well Calvin could find the actual culprit. The first, the number of suspects collected, show how well the suspect collection routine discriminated among the circuit's components. The next value, number of hypotheses checked, shows how many hypotheses Calvin checked during the diagnosis. This higher this number is, the more hypothesis Calvin had to run to determine if a suspect could cause the reported outputs. The third number is how many hypotheses Calvin found that could cause the reported outputs. Since one assumption was that there was only a single fault, ideally this value should be one. The last three numbers give an idea of how many computations that Calvin must do. Since these numbers are closely related to the circuit's input values and number of hypotheses that Calvin test, they are not important. They were used mainly as debugging tools during implementation.

For a detailed example, consider the Full-Adder shown in Figure 31. While the full-adder source was being parsed, Calvin generated mcode-blocks that would simulate the correctly-operating version of each process in the full-adder. After each block was created, it was sent to a hypothesis generating module. This module generated additional mcode-blocks to simulate the errors that Calvin was programmed to check. These were gathered and placed in a **behave** object that represented the process. A sample test file is shown in Figure 34. The top eight lines simulated correct behavior of the circuit. The following lines simulated the circuit after certain faults had been introduced. Calvin was run, showing the values Calvin thought should be at the output, as well as the possible faults Calvin found. For the correct outputs, Calvin reported that its outputs values matched those reported. Calvin then stated that no errors were found.

The first fault introduced was the output of I082 stuck high. In this case, no matter what the inputs are, the carry-out will always be high. The first set of input had all inputs low. In this case both outputs also should be low, which Calvin also determined. Since the output values as determined by Calvin did not match those supplied by the test file, Calvin attempted to find the fault. The first step was to collect the possible suspects. Working upstream from I082 (the carry-out OR-gate), Calvin determined that I082, I081, I080, and I080 could be at fault. The duplicate I080 component came from looking at the inputs of I081 as well as I082. An improved routine that stopped the depth-first search when a previously-found suspect was encountered eliminated the duplicate I080.

Calvin simulated each hypothesis for a suspect component, starting with I082. When Calvin was parsing the VHDL code for an OR-gate, it determined that there were six possible problems that could happen to the gate: output number 0 stuck high/low, input number 0 stuck high/low, and input number 1 stuck high/low. Calvin hypothesized each of these problems, and found that the following could cause the reported values:

- output #0 stuck hi,

| Inputs | | | Outputs | | Comments |
|---|---|---|---|---|---|
| I051 (X) | I052 (Y) | I053 (Cin) | I054 (Sum) | I055 (Cout) | |
| 0 | 0 | 0 | 0 | 0 | Correct operation |
| 0 | 0 | 1 | 1 | 0 | |
| 0 | 1 | 0 | 1 | 0 | |
| 0 | 1 | 1 | 0 | 1 | |
| 1 | 0 | 0 | 1 | 0 | |
| 1 | 0 | 1 | 0 | 1 | |
| 1 | 1 | 0 | 0 | 1 | |
| 1 | 1 | 1 | 1 | 1 | |
| 0 | 0 | 0 | 0 | 1 | I082 Out stuck high |
| 0 | 1 | 1 | 0 | 1 | I082 Out stuck high |

Figure 34. Test Data for FULLADD.VHD

| |
|---|
| Output #0 stuck hi at 82 |
| Input #0 stuck hi at 82 |
| Input #1 stuck hi at 82 |
| Output #1 stuck hi at 81 |
| Output #1 stuck hi at 80 |

Figure 35. Faults Found in FULLADD.VHD With I082's Output Tied High

- input #0 stuck hi,
- input #1 stuck hi.

Calvin did the same with the rest of the suspects. A complete list of possible faults Calvin found is in Figure 35. Note this does include the introduced fault, I082 output stuck high. The half-adders each had 8 hypothesis (each of the four ports stuck high and stuck low), and the OR-gate had 6 (three ports stuck high/low). This meant Calvin checked a total of 22 hypotheses.

In the next example, the same fault (I082 output stuck high) was kept in the simulated "real" circuit. But this time the inputs were 0, 1 and 1. Calvin simulated the circuit and found its outputs the same as those reported to Calvin. Calvin decided there was no problem with the circuit.

A more complex example uses the ALU shown in Figure 32. Sample inputs to the circuit, along with the outputs from a "real" circuit are in Figure 36. The circuit without the probes was used first. To do this, the output signals I710-I712 and I810-I812 were commented out, and internal signals with the same name were declared. This VHDL code is in Appendix B.2. Values for the

| Inputs | | | | | | | Outputs | | |
|---|---|---|---|---|---|---|---|---|---|
| I595 S | I510 A0 | 511 A1 | 512 A2 | I520 B0 | I521 B1 | I522 B2 | I530 Z0 | I531 Z1 | I532 Z2 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

Figure 36. Test Data for ALU.VHD

| |
|---|
| Output #0 stuck low at 605 |
| Input #0 stuck low at 605 |
| Output #0 stuck low at 603 |
| Input #0 stuck low at 603 |
| Input #1 stuck low at 603 |
| Output #0 stuck low at 601 |
| Input #0 stuck low at 601 |
| Input #1 stuck low at 601 |

Figure 37. Faults Found in ALU.VHD With I601's Output Tied Low

correctly working "real" circuit are at the top of the table, with simulated fault conditions and their output values below. The first fault introduced was shorting output of gate i601 low. With all inputs high, and the AND operation selected, Calvin determined that all the outputs also should have been high. Comparing against those supplied by the "real" circuit, Calvin found one of the components tied to the I530 output was at fault. The suspects were I605, I604, I603, I606, I602, and I601. Since none of the other components affect the output I503, Calvin did not hypothesize any of others. After checking out all the hypotheses for the I601-I606, Calvin came up with a list of possible suspects. These are in Figure 37. Calvin did find the fault, I601's output stuck low, although it also came up with seven others.

This case was re-run using the ALU circuit with probes (Appendix B.3). The same inputs and outputs were used, with the additional values for the probes supplied. These values are in Figure 38. With the additional information, Calvin was able to narrow it down to three suspects, which are listed in Figure 39. Since I710 along with I530 was low, the hypotheses for I603 and I605 were ruled out. No hypothesis for these components could account for the '1' value at I710.

| Inputs | | | | | | | Outputs | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I595 S | I510 A0 | 511 A1 | 512 A2 | I520 B0 | I521 B1 | I522 B2 | I530 Z0 | I531 Z1 | I532 Z2 | I710 YA0 | I810 YO0 | I711 YA1 | I811 YO1 | I712 YA2 | I812 YO2 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

Figure 38. Test Data for ALU.VHD with Sensors

```
Output #0 stuck low at 601
 Input #0 stuck low at 601
 Input #1 stuck low at 601
```

Figure 39. Faults Found in ALU.VHD With Probes

One characteristic of combinatorial digital logic is that most faults cannot be found using one set of test data. For example, if an input was stuck low, and the actual signal value was low, the circuit would operate correctly. The circuit's sensors would show the correct result for this set of commands. The fault will only become apparent when the signal value is high.

## *4.2 Improvements*

*4.2.1 Improving the Hypothesis Generator* Calvin currently takes all components upstream from the incorrect output and places them in the suspect queue. Calvin could use knowledge of boolean algebra to better select candidate suspects. For example, consider one bit of the ALU shown in Figure ref401. Assume that both data inputs are high, and the select line is high for an OR operation. When simulating the circuit's processes, Calvin would place "reasons" for a signal's value in the activation record along with the signal's new value. These could then be placed in the **Behave** object. For OR gate I605, the value on output signal I530 would be 1 because of input signal I004's 1 state. Now, suppose that the signal I530 is found to be low instead of the simulated high state. Since I530 was caused by I003, and OR gate I605 was assumed to be working, Calvin would not have to consider components upstream from the other input, I003. (11:394-396)

*4.2.2 Probing* By bringing out internal signals to sensors, Calvin reduced the number of suspect hypotheses dramatically. This is, in effect, inserting additional probes into the circuit.

Calvin contains component connection data within its **Behave** and **Signal** objects. Reasoning with this information, an additional module can suggest possible signal lines to monitor.

An easy way would be to reason from the topology of the circuit. A simple way would be to ask for the value of the signal immediately upstream from the incorrect output. If this matches the simulated signal's value, the component downstream must be the culprit. If not, continue upstream. Calvin can support this technique, with its **driver_bi** field in the **Signal** object.

A more efficient technique, described by Davis and Hamscher, would be to perform what is in effect a binary search. Examining the connection diagram of the circuit, Calvin could select a signal that roughly divides the circuit into two section. If that signal's actual value still does not match the simulated value, all components downstream from that signal are exonerated. This can continue, ideally splitting the suspect components into two groups each time, until a single suspect is found. (11:410-411)

*4.2.3 Extending the VHDL language* As with all models, VHDL has some limitations. VHDL lacks a way to describe easily the physical layout of a system. Examining the VHDL source for the ALU in Appendix B.2, one cannot determine if the OR gates are on a common IC. This prevents easily determining faults such as solder bridges, or broken power pins. Although a single fault, the effects might be seen in two entirely separate parts of the circuit.

One way of extending the language for diagnostic purposes would be adding commands for explicitly stating what certain errors might do. The C language handles implementation specific features by the **pragma** keyword; in VHDL this might be handled with any new keywords embedded within comment lines. These additions could link any processes that describe error conditions to those that describe proper behavior. In Calvin, this would require additions only to the parser; the errant processes would be linked to proper behaviors.

*4.2.4 Interfacing with an Expert System* Calvin, as it exists now, only has the rudiments of AI behavior. A rule-based expert system could be interfaced with Calvin, turning it into a hybrid model-based/rule-based system. Obvious places that would benefit would be the fault-hypothesis and the suspect selection.

Knowledge that certain subcomponents have a high failure rate could be placed in the source. A good place would be in library packages that are reused. With rules based on this information, the suspect queue could be reordered to place these components nearer the front Although this would not improve the current implementation, a real-time version might be able to come up with a "best guess" if there is not enough time to complete the algorithm.

## *4.3 Summary*

In this chapter I discussed three VHDL descriptions that Calvin used. Then I discussed ways in which the current Calvin program can be extended and made more powerful. In the next chapter I will summarize the thesis and discuss my recommendations for future work.

# V. Observations and Recommendations

## 5.1 Review

In Chapter 1 is a review of the problem that this thesis attacks. Chapter 2 records a review of some model-based diagnostic techniques, including Scarl's "Full consistency" and Dries' "Diagnose" algorithms. Chapter 3 contains a description of Calvin, including the internal structures and algorithms. A description of the test programs is in Chapter 4, along with ideas on how Calvin can be extended.

## 5.2 Accomplishments

In this thesis effort I designed Calvin, a model-based diagnostic system that used VHDL to describe the model. Calvin is a starting point that can be used for more powerful model-based paradigms. Some of my accomplishments during this effort:

- VHDL can be used to describe the model for a model-based diagnostic technique. VHDL was designed to be a description language, and future Air Force contracts mandate its use.
- The goal of creating Calvin was to develop a VHDL system that can be used for implementing model-based diagnostic techniques. I did this by:
  - Implementing a VHDL parser that generated a representation of a system from a VHDL source file. This representation included all the necessary information for reasoning about the structure of the components in the system, as well as the behavior of those components.
  - Implementing a VHDL simulator that could take the representation from the parser and a set of input values, and simulate the system generating output values.
  - Creating routines that could modify the parsed representation so that errors in the behavior of the original circuit can be simulated.

- I implemented a version of Scarl's "full consistency" algorithm that was based on Dries' "Diagnose" algorithm.

### *5.3 Recommendations*

Based on my experiences, the following is a list of recommendations for future work:

- Expand the implemented VHDL language. The implemented subset is limited to only logical operations. Expanding the VHDL constructs that Calvin can recognize will allow much more complex systems. These include those whose behavior is described by loops or conditional statements. Implementing the VHDL **TYPE** command would allow multi-value logic systems, such as an "OFF" state. Diagnosing analog systems would require floating point arithmetic. Since the parser ignores unknown constructs, there should not have to be much rewriting of existing code.
- Implement the probing improvements that Chapter 4 describes. These should be the easiest to add. I have already manually placed probes in the source code, so Calvin should not need new data structures.
- Improve the hypothesis generator. Here should be the easiest place to put "real AI" into Calvin. Calvin already has the data structures for reasoning on the structure of the circuit. The hardest problem may be reasoning from the VHDL processes.
- Extend the diagnostic algorithm. The current algorithm only allows non-time-dependent systems to be diagnosed. The VHDL model contains time information, so there should be a way to reason on systems that have "memory," such as sequential systems. One way may be to maintain the list of suspects, and use that information when new inputs enter the test system. The new values should affirm certain hypotheses and reject others.

## *5.4 Summary*

To perform model-based diagnostics, there must be some model to reason from. To keep from having to build a new diagnostic system for each new product, a language can be used for the model of the product. One approach may be to create a special-purpose language for describing the model; however, there are already description languages in existence. One of these languages is VHDL, a VHSIC hardware description language. Calvin uses VHDL as a model description language.

In artificial intelligence applications, sometimes it seems as if very little of the project involves any "artificial intelligence." Instead, most of the effort is in creating a platform for the application. This is also true for using VHDL for model-based diagnostics. Most of my time was spent getting the simulator and parser portions of Calvin working. A lot of the rest was taken up interfacing Dries' *Diagnose* algorithm with the simulator and parser.

With Calvin, I have created an important framework that future researchers can easily extend. Future work should be on extending Calvin's diagnostic and simulation routines.

# Appendix A. *Supported VHDL Grammar*

```
entity_declaration ::=
    ENTITY identifier IS
        entity_header
        entity_declarative_part
    END;

entity_header ::=
    port_clause

port_clause ::=
    port ( formal_port_list );

formal_port_list ::=
    [format_port_element]

formal_port_element ::=
    SIGNAL identifier_list : mode
    | SIGNAL identifier_list : mode ; formal_port_element

mode ::=
    IN
    | OUT

architecture_body ::=
    ARCHITECTURE identifier OF name IS
        architecture_declarative_part
    BEGIN
        architecture_statement_part
    END;

architecture_declarative_part ::=
    [block_declarative_item]

block_declarative_item ::=
    signal_declaration
    | component_declaration

architecture_statement_part ::=
    [concurrent_statement]

configuration_declaration ::=
    CONFIGURATION identifier OF name IS
        block_configuration
    END;
```

```
block_configuration ::=
    FOR block_specification
        use_clause
        configuration_item
    END FOR;

block_specification ::=
    identifier

use_clause ::=
    USE identifier[,identifier];

configuration_item ::=
    component_configuration

component_configuration ::=
    FOR instantiation_list : identifier
    use_binding_indication
    block_configuration
    END FOR;

use_binding_indication ::=
    USE binding_indication;

binding_indication ::=
    entity_aspect

entity_aspect ::=
    ENTITY identifier ( identifier )

signal_declaration ::=
    SIGNAL identifier_list : BIT

component_declaration ::=
    COMPONENT identifier
        port_local_port_list
    END COMPONENT;

port_local_port_list ::=
    ( local_port_list );

local_port_list ::=
    identifier_list : local_port_mode BIT
    | identifier_list : local_port_mode BIT;
    local_port_list

local_port_mode ::=
    IN
    | OUT
```

concurrent_statement ::=
    process_statement

process_statement ::=
    **PROCESS**
    **BEGIN**
        sequence_of_statements
    **END PROCESS;**

sequence_of_statements ::=
    {sequential_statement}

sequential_statement ::=
    signal_assignment_statement

signal_assignment_statement ::=
    target ¡= waveform;

terget ::=
    identifier

waveform ::=
    expression **AFTER** expression

expression ::=
    relation_and_relation
    | relation_or_relation
    | relation_nand_nor_relation
    | relation_xor_relation

relation_and_relation ::=
    relation **AND** relation

relation_or_relation ::=
    relation **OR** relation

relation_nand_nor_relation ::=
    relation
    | relation **NAND** relation
    | relation **NOR** relation

relation_xor_relation ::=
    relation **XOR** relation

relation ::=
    simple_expression

```
simple_expression ::=
    primary

primary ::=
    literal
    | identifier

literal ::=
    numeric_literal

numeric_literal ::=
    decimal_int

decimal_int ::=
    [digit]
```

# Appendix B. *VHDL Source Code*

## *B.1 Full-Adder*

This section contains the VHDL source for a full-adder. This file was used by Calvin.

```
--
-- One-bit full-adder
--
-- Consists of 2 half-adders and an OR gate
--
-- X + Y + Cin = Z + Cout
--
-- This full-adder is used in the four-bit adder
--
-------------------------
-------- OR Gate ---------
entity i015 is
port(
   i011: in  Bit;
   i012: in  bit;
   i013: out bit
);
end;

architecture i025 of i015 is
begin
   process
   begin
      i013 <= i011 or i012 after 5;
   end process;
end i010;
-------------------------
------ Half adder --------
entity i010 is
port(
   i011: in  Bit;
   i012: in  bit,
   i013: out bit;
   i014: out bit
);
end;

architecture i020 of i010 is
begin
   process
   begin
      i013 <= i011 xor i012 after 5;
      i014 <= i011 and i012 after 5;
   end process;
```

```
end i010;
--------------------------
------ Full Adder --------
entity i050 is
port(
   i051,i052,i053:in bit;
   i054,i055:out bit
);
end;

architecture i060 of i050 is

signal i090:bit;
signal i091:bit;
signal i092:bit;
component i010
   port(
      i011: in  Bit;
      i012: in  bit;
      i013: out bit;
      i014: out bit
   );
end component;
component i030
   port(
      i011,i012:in bit;
      i013:out bit
   );
end component;

begin
   i080:i010
      port map(
         i011 => i051,
         i012 => i052,
         i013 => i090,
         i014 => i091 );

   i081:i010
      port map(
         i011 => i090,
         i012 => i053,
         i013 => i054,
         i014 => i092 );
   i082:i030
      port map(
         i011 => i091,
         i012 => i092,
         i013 => i055 );
end;
```

```
-------------------------
------- Circuit ----------
configuration i099 of i050 is
  for i060
    for i080,i081:i010 use entity i010(i020);
    end for;
    for i082:i030 use entity i015(i025);
    end for;
  end for;
end;
-------------------------
```

*B.2 Two Operation ALU*

This section contains the VHDL source for a two-operation AND/OR ALU The code to simulate probes placed in the circuit are commented out in this code.

```
--
-- Three-bit, Two-operation ALU
-- Performs AND or OR function of 2 three-bit values
--
-- If S=1, A2A1A0 AND B2B1B0 = Z2Z1Z0
-- If S=0, A2A1A0 OR  B2B1B0 = Z2Z1Z0
--
--
-- This example has the probes inserted at the outputs
-- of the AND/OR functions commented out.
--------------------------
-------- OR Gate ---------
entity i200 is
port(
   i201: in  Bit;
   i202: in  bit;
   i203: out bit
);
end;
architecture i299 of i200 is
begin
   process
   begin
      i203 <= i201 or i202 after 5;
   end process;
end;
--------------------------
-------- AND Gate ---------
entity i100 is
port(
   i101: in  Bit;
   i102: in  bit;
   i103: out bit
);
end;
architecture i199 of i100 is
begin
   process
   begin
      i103 <= i101 and i102 after 5;
   end process;
end;
--------------------------
-------- INVGate ---------
entity i300 is
port(
   i301: in  Bit;
```

```
   i302: out bit
);
end;
architecture i399 of i300 is
begin
   process
   begin
      i302 <= not i301 after 5;
   end process;
end;
-------------------------

entity i500 is
   port(
      i510  : in  bit; -- A
      i511  : in  bit; -- A
      i512  : in  bit; -- A
      i520  : in  bit; -- B
      i521  : in  bit; -- B
      i522  : in  bit; -- B
      i595  : in  bit; -- s0
      i530  : out bit; -- Z
      i531  : out bit; -- Z
      i532  : out bit -- Z
--
-- The following are the commented-out probes
--
--      i710  : out bit; -- Y0AND
--      i711  : out bit; -- Y1AND
--      i712  : out bit; -- Y1AND
--      i810  : out bit; -- Y0OR
--      i811  : out bit; -- Y1OR
--      i812  : out bit  -- Y1OR
      );
end;

architecture i599 of i500 is

   component i100
   port( i101,
         i102   :  In    Bit;
         i103   :  out   Bit );
   end component;

   component i200
   port( i201,
         i202   :  In    Bit;
         i203   :  out   Bit );
   end component;

   component i300
```

```
   port( i301   :  In    Bit;
         i302   :  Out   Bit );
   end component;

signal
   i000,
   i001,i003,i004,
   i011,i013,i014,
   i021,i023,i024
   : bit;

--
-- The commented-out probes have been replaced by
-- these internal signals
--
signal i710,i711,i712 : bit;
signal i810,i811,i812 : bit;

begin
   -- Control line inverter
   i606: i300 port map( i301=>i595, i302=>i000 );

   -- Bit 0
   i601: i100 port map( i101=>i510, i102=>i520, i103=>i710 );
   i602: i200 port map( i201=>i510, i202=>i520, i203=>i810 );
   i603: i100 port map( i101=>i710, i102=>i000, i103=>i003 );
   i604: i100 port map( i101=>i810, i102=>i595, i103=>i004 );
   i605: i200 port map( i201=>i003, i202=>i004, i203=>i530 );

   -- Bit 1
   i611: i100 port map( i101=>i511, i102=>i521, i103=>i711 );
   i612: i200 port map( i201=>i511, i202=>i521, i203=>i811 );
   i613: i100 port map( i101=>i711, i102=>i000, i103=>i013 );
   i614: i100 port map( i101=>i811, i102=>i595, i103=>i014 );
   i615: i200 port map( i201=>i013, i202=>i014, i203=>i531 );

   -- Bit 2
   i621: i100 port map( i101=>i512, i102=>i522, i103=>i712 );
   i622: i200 port map( i201=>i512, i202=>i522, i203=>i812 );
   i623: i100 port map( i101=>i712, i102=>i000, i103=>i023 );
   i624: i100 port map( i101=>i812, i102=>i595, i103=>i024 );
   i625: i200 port map( i201=>i023, i202=>i024, i203=>i532 );

end;
-------------------------
------- Circuit ----------
configuration i000 of i500 is
  for i599
    -- AND gates
    for i601,i603,i604:i100 use entity i100(i199);
    end for;
```

```
    for i611,i613,i614:i100 use entity i100(i199);
    end for;

    for i621,i623,i624:i100 use entity i100(i199);
    end for;

    -- OR  gates
    for i602,i605:i200 use entity i200(i299);
    end for;

    for i612,i615:i200 use entity i200(i299);
    end for;

    for i622,i625:i200 use entity i200(i299);
    end for;

    -- INV gates
    for i606:i300 use entity i300(i399);
    end for;

  end for;
end;
--------------------------
```

*B.3 Two Operation ALU with Probes*

This section contains the VHDL source for a two-operation AND/OR ALU This code contains the probes placed in the circuit.

```
--
-- Three-bit, Two-operation ALU
-- Performs AND or OR function of 2 three-bit values
--
-- If S=1, A2A1A0 AND B2B1B0 = Z2Z1Z0
-- If S=0, A2A1A0 OR  B2B1B0 = Z2Z1Z0
--
--
-- This example has the probes inserted at the outputs
-- of the AND/OR functions. These bring the results of  both
-- functions to sensors.
-------------------------
-------- OR Gate ---------
entity i200 is
port(
   i201: in  Bit;
   i202: in  bit;
   i203: out bit
);
end;
architecture i299 of i200 is
begin
   process
   begin
      i203 <= i201 or i202 after 5;
   end process;
end;
-------------------------
-------- AND Gate ---------
entity i100 is
port(
   i101: in  Bit;
   i102: in  bit;
   i103: out bit
);
end;
architecture i199 of i100 is
begin
   process
   begin
      i103 <= i101 and i102 after 5;
   end process;
end;
-------------------------
-------- INVGate ---------
entity i300 is
port(
```

```
   i301: in  Bit;
   i302: out bit
);
end;
architecture i399 of i300 is
begin
   process
   begin
      i302 <= not i301 after 5;
   end process;
end;
--------------------------

entity i500 is
   port(
      i510  : in  bit; -- A
      i511  : in  bit; -- A
      i512  : in  bit; -- A
      i520  : in  bit; -- B
      i521  : in  bit; -- B
      i522  : in  bit; -- B
      i595  : in  bit; -- s0
      i530  : out bit; -- Z
      i531  : out bit; -- Z
      i532  : out bit; -- Z
--
-- These output signals are the probes
--
      i710  : out bit; -- Y0AND
      i711  : out bit; -- Y1AND
      i712  : out bit; -- Y1AND
      i810  : out bit; -- Y0OR
      i811  : out bit; -- Y1OR
      i812  : out bit  -- Y1OR
      );
end;

architecture i599 of i500 is

   component i100
   port( i101,
         i102   :  In    Bit;
         i103   :  out   Bit );
   end component;

   component i200
   port( i201,
         i202   :  In    Bit;
         i203   :  out   Bit );
   end component;
```

```
   component i300
   port( i301    :  In    Bit;
         i302    :  Out   Bit );
   end component;

signal
   i000,
   i001,i003,i004,
   i011,i013,i014,
   i021,i023,i024
   : bit;

begin
   -- Control line inverter
   i606: i300 port map( i301=>i595, i302=>i000 );

   -- Bit 0
   i601: i100 port map( i101=>i510, i102=>i520, i103=>i710 );
   i602: i200 port map( i201=>i510, i202=>i520, i203=>i810 );
   i603: i100 port map( i101=>i710, i102=>i000, i103=>i003 );
   i604: i100 port map( i101=>i810, i102=>i595, i103=>i004 );
   i605: i200 port map( i201=>i003, i202=>i004, i203=>i530 );

   -- Bit 1
   i611: i100 port map( i101=>i511, i102=>i521, i103=>i711 );
   i612: i200 port map( i201=>i511, i202=>i521, i203=>i811 );
   i613: i100 port map( i101=>i711, i102=>i000, i103=>i013 );
   i614: i100 port map( i101=>i811, i102=>i595, i103=>i014 );
   i615: i200 port map( i201=>i013, i202=>i014, i203=>i531 );

   -- Bit 2
   i621: i100 port map( i101=>i512, i102=>i522, i103=>i712 );
   i622: i200 port map( i201=>i512, i202=>i522, i203=>i812 );
   i623: i100 port map( i101=>i712, i102=>i000, i103=>i023 );
   i624: i100 port map( i101=>i812, i102=>i595, i103=>i024 );
   i625: i200 port map( i201=>i023, i202=>i024, i203=>i532 );

end;
-------------------------
------- Circuit ----------
configuration i000 of i500 is
  for i599
    -- AND gates
    for i601,i603,i604:i100 use entity i100(i199);
    end for;

    for i611,i613,i614:i100 use entity i100(i199);
    end for;

    for i621,i623,i624:i100 use entity i100(i199);
    end for;
```

```
    -- OR  gates
    for i602,i605:i200 use entity i200(i299);
    end for;

    for i612,i615:i200 use entity i200(i299);
    end for;

    for i622,i625:i200 use entity i200(i299);
    end for;

    -- INV gates
    for i606:i300 use entity i300(i399);
    end for;

  end for;
end;
-------------------------
```

### *B.4 Four-bit Adder*

This section contains the VHDL source for a four-bit adder. This code was used by Calvin.

```
--
-- Four-bit Adder
--
-- Consists of 4 full-adders in cascade
--
-- X3X2X1X0 + Y3Y2Y1Y0 + Cin = Z3Z2Z1Z0 + Cout
--
--------------------------
-------- OR Gate ---------
entity i015 is
port(
   i011: in  Bit;
   i012: in  bit;
   i013: out bit
);
end;

architecture i025 of i015 is
begin
   process
   begin
      i013 <= i011 or i012 after 5;
   end process;
end i010;
--------------------------
------ Half adder --------
entity i010 is
port(
   i011: in  Bit;
   i012: in  bit;
   i013: out bit;
   i014: out bit
);
end;

architecture i020 of i010 is
begin
   process
   begin
      i013 <= i011 xor i012 after 5;
      i014 <= i011 and i012 after 5;
   end process;
end i020;
--------------------------
------ Full Adder --------
entity i050 is
port(
   i100  : in  bit;  -- Cin
```

```
   i110,             -- X0
   i111,             -- X1
   i112,             -- X2
   i113  : in  bit;  -- X3
   i120,             -- Y0
   i121,             -- Y1
   i122,             -- Y2
   i123  : in  bit;  -- Y3
   i130,             -- Z0
   i131,             -- Z1
   i132,             -- Z2
   i133  : out bit;  -- Z3
   i140,             -- cout0
   i141,             -- cout1
   i142  : out bit;  -- cout2
   i143  : out bit   -- Cout
);
end;

architecture i060 of i050 is


signal i200,i201,i202:bit;
signal i210,i211,i212:bit;
signal i220,i221,i222:bit;
signal i230,i231,i232:bit;

component i010
   port(
      i011: in  Bit;
      i012: in  bit;
      i013: out bit;
      i014: out bit
   );
end component;
component i030
   port(
      i011,i012:in bit;
      i013:out bit
   );
end component;

begin
-- Bit 0
   i500:i010
      port map(
         i011 => i110,
         i012 => i120,
         i013 => i200,
         i014 => i201 );
```

```
   i501:i010
      port map(
         i011 => i200,
         i012 => i100,
         i013 => i130,
         i014 => i202 );

   i502:i030
      port map(
         i011 => i202,
         i012 => i201,
         i013 => i140 );

-- Bit 1
   i510:i010
      port map(
         i011 => i111,
         i012 => i121,
         i013 => i210,
         i014 => i211 );

   i511:i010
      port map(
         i011 => i210,
         i012 => i140,
         i013 => i131,
         i014 => i212 );

   i512:i030
      port map(
         i011 => i212,
         i012 => i211,
         i013 => i141 );
-- Bit 2
   i520:i010
      port map(
         i011 => i112,
         i012 => i122,
         i013 => i220,
         i014 => i221 );

   i521:i010
      port map(
         i011 => i220,
         i012 => i141,
         i013 => i132,
         i014 => i222 );

   i522:i030
      port map(
         i011 => i222,
```

```
         i012 => i221,
         i013 => i142 );
-- Bit 3
   i530:i010
      port map(
         i011 => i113,
         i012 => i123,
         i013 => i230,
         i014 => i231 );

   i531:i010
      port map(
         i011 => i230,
         i012 => i142,
         i013 => i133,
         i014 => i232 );

   i532:i030
      port map(
         i011 => i232,
         i012 => i231,
         i013 => i143 );
end;

-------------------------
------- Circuit ----------
configuration i099 of i050 is
  for i060
    for i500,i501:i010 use entity i010(i020);
    end for;
    for i502:i030 use entity i015(i025);
    end for;

    for i510,i511:i010 use entity i010(i020);
    end for;
    for i512:i030 use entity i015(i025);
    end for;

    for i520,i521:i010 use entity i010(i020);
    end for;
    for i522:i030 use entity i015(i025);
    end for;

    for i530,i531:i010 use entity i010(i020);
    end for;
    for i532:i030 use entity i015(i025);
    end for;
  end for;
end;
-------------------------
```

### *B.5 Five-bit 2's Compliment ALU*

This section contains the VHDL source for a five-bit 2's Compliment ALU. The ALU performs addition and subtraction. This section is included as an additional example that can be used with Calvin.

```
-- This code describes a 5 bit 2's bit ALU.
-- Operations include add and subtract of two
-- 5-bit 2's compliment numbers.
--
-------------------------
-------- OR Gate ---------
entity i100 is
port(
   i101: in  Bit;
   i102: in  bit;
   i103: out bit
);
end;
architecture i199 of i100 is
begin
   process
   begin
      i103 <= i101 or i102 after 5;
   end process;
end;
-------------------------
-------- AND Gate ---------
entity i200 is
port(
   i201: in  Bit;
   i202: in  bit;
   i203: out bit
);
end;
architecture i299 of i200 is
begin
   process
   begin
      i203 <= i201 and i202 after 5;
   end process;
end;
-------------------------
-------- INVGate ---------
entity i300 is
port(
   i301: in  Bit;
   i302: out bit
);
end;
architecture i399 of i300 is
begin
```

```
   process
   begin
      i302 <= not i301 after 5;
   end process;
end;
--------------------------
--------------------------
-- FULL_ADDER
entity i400 is
port(
   i401: in  bit;  -- x
   i402: in  bit;  -- y
   i403: in  bit;  -- Cin
   i404: out bit;  -- Sum
   i405: out bit   -- Cout
);
end;
architecture i499 of i400 is
begin
   process
   begin
      i404 <= i401 xor i402 xor i403 after 5;
      i405 <= (i401 and i402) or
              (i403 and i401) or
              (i403 and i402) after 5;
   end process;
end;
--------------------------
--------------------------
entity i500 is
   port(
      i510  : in  bit; -- X
      i511  : in  bit;
      i512  : in  bit;
      i513  : in  bit;
      i514  : in  bit;
      i520  : in  bit; -- Y
      i521  : in  bit;
      i522  : in  bit;
      i523  : in  bit;
      i524  : in  bit;
      i544  : in  bit; -- add
      i545  : in  bit; -- sub
      i530  : out bit; -- S
      i531  : out bit;
      i532  : out bit;
      i533  : out bit;
      i534  : out bit
      );
end;
architecture i599 of i500 is
```

```
   component i200
   port( i201,
         i202   :  In    Bit;
         i203   :  out   Bit );
   end component;

   compcnent i100
   port( i101,
         i102   :  In    Bit;
         i103   :  out   Bit );
   end component;

   component i300
   port( i301   :  In    Bit;
         i302   :  Out   Bit );
   end component;

   component i400
   port(
      i401: in  bit;  -- x
      i402: in  bit;  -- y
      i403: in  bit;  -- Cin
      i404: out bit;  -- Sum
      i405: out bit   -- Cout
      );
   end component;

   signal   i998,
            i001, i002, i003, i004, i005, i900,
            i011, i012, i013, i014, i015, i901,
            i021, i022, i023, i024, i025, i902,
            i031, i032, i033, i034, i035, i903,
            i041, i042, i043, i044, i045, i904  : Bit;

begin
   i701 :  i100 port map( i101=>i544, i102=>i545, i103=>i998 );
   -- Bit add/sub
   -- Bit 0
   i710 :  i300 port map( i301=>i520, i302=>i001 );
   i711 :  i200 port map( i201=>i545, i202=>i001, i203=>i002 );
   i712 :  i200 port map( i201=>i544, i202=>i520, i203=>i003 );
   i713 :  i200 port map( i201=>i510, i202=>i998, i203=>i004 );
   i714 :  i100 port map( i101=>i003, i102=>i002, i103=>i005 );
   i715 :  i400 port map( i401=>i004, i402=>i005, i403=>i545,
           i404=>i530, i405=>i900 );
   -- Bit 1
   i720 :  i300 port map( i301=>i521, i302=>i011 );
   i721 :  i200 port map( i201=>i545, i202=>i011, i203=>i012 );
   i722 :  i200 port map( i201=>i544, i202=>i521, i203=>i013 );
   i723 :  i200 port map( i201=>i511, i202=>i998, i203=>i014 );
```

```
   i724 :  i100 port map( i101=>i013, i102=>i012, i103=>i015 );
   i725 :  i400 port map( i401=>i014, i402=>i015, i403=>i900,
           i404=>i531, i405=>i901 );
   -- Bit 2
   i730 :  i300 port map( i301=>i522, i302=>i021 );
   i731 :  i200 port map( i201=>i021, i202=>i545, i203=>i022 );
   i732 :  i200 port map( i201=>i544, i202=>i522, i203=>i023 );
   i733 :  i200 port map( i201=>i512, i202=>i998, i203=>i024 );
   i734 :  i100 port map( i101=>i023, i102=>i022, i103=>i025 );
   i735 :  i400 port map( i401=>i024, i402=>i025, i403=>i901,
           i404=>i532, i405=>i902 );
   -- Bit 3
   i740 :  i300 port map( i301=>i523, i302=>i031 );
   i741 :  i200 port map( i201=>i031, i202=>i545, i203=>i032 );
   i742 :  i200 port map( i201=>i544, i202=>i523, i203=>i033 );
   i743 :  i200 port map( i201=>i513, i202=>i998, i203=>i034 );
   i744 :  i100 port map( i101=>i033, i102=>i032, i103=>i035 );
   i745 :  i400 port map( i401=>i034, i402=>i035, i403=>i902,
           i404=>i533, i405=>i903 );
   -- Bit 4
   i750 :  i300 port map( i301=>i524, i302=>i041 );
   i751 :  i200 port map( i201=>i041, i202=>i545, i203=>i042 );
   i752 :  i200 port map( i201=>i544, i202=>i524, i203=>i043 );
   i753 :  i200 port map( i201=>i514, i202=>i998, i203=>i044 );
   i754 :  i100 port map( i101=>i043, i102=>i042, i103=>i045 );
   i755 :  i400 port map( i401=>i044, i402=>i045, i403=>i903,
           i404=>i534, i405=>i904 );
--Done
end;
---------------------------
------- Circuit ----------
configuration i000 of i500 is
  for i599
    for i701:i100 use entity i100(i199);
    end for;
    for i710,i720,i730,i740,i750:i300 use entity i300(i399);
    end for;
    for i711,i712,i713,i721,
        i722,i723,i731,i732,
        i733,i741,i742,i743,
        i751,i752,i753:i200 use entity i200(i299);
    end for;
    for i714,i724,i734,i744,i754:i100 use entity i100(i199);
    end for;
    for i715,i725,i735,i745,i755:i400 use entity i400(i499);
    end for;
  end for;
end;
---------------------------
```

# Appendix C. *FLEX modifications*

Before the Borland C++ 3.1 compiler could compile the output of FLEX, some changes had to be made to FLEX.SKL file. This file forms the skeleton of the FLEX output file. The following changes were made:

1. Remove line 23 - "#include <osfcn.h>"
2. Remove line 33 - "#ifdef __STDC__"
3. Remove line 46 - "#endif __STDC__"

No changes were required for Bison.

# Appendix D. *Compiler-compiler Source Code*

## *D.1 Overview*

This module contains code for the compiler-compilers Bison and FLEX. The module **UV** describes the grammar for VHDL. The individual VHDL tokens are parsed by FLEX according to the module **UV.LEX**. These tokens are then parsed according to the VHDL grammar. As soon as a construct has been recognized, the appropriate parser module is called to fill in the data. These modules are described in Appendix E.

## *D.2 UV*

```
/*

   FILE: UV

   This is the parser file for Calvin.
   The simulator objects are built up by calling external C/C++functions
   defined in the Parser modules.

*/
/*****************************************************************************
**
** Portions of the following code was extracted from
**    LALR(1) grammar for ANSI Ada (public domain)
**    by Herman Fischer
**    adapted by: Gerry Fisher & Philippe Charles
**
**  VHDL source for yacc
** syntax analysis with error recovery
** symbol table
** memory allocation
** no code generation
** shift/reduce conflicts: 1
**
**
** Symbol conventions used:
**    [foo]        is denoted  _foo_
**    {foo}        is denoted  __foo__
**    {, foo }     is denoted  ___foo__
**    foo_bar      is a single nonterminal
```

```
**    FOO__bar    is a nonterminal where the keyword FOO is
**                   followed by a nonterminal bar
**
*************************************************************************/
// History of original VHDL grammar
/************************************************************************
*
* Date: 19 Feb, 1990 S. Datta, Univ of Cincinnati
*
*This file currently contains 3 shift/reduce and 3 reduce/reduce conflicts:
*
*Shift/reduce conflicts:
*
* 1.   name -> simple_name
*and    .architecture_identifier. -> LeftParen simple_name RightParen_ERR
*cause 1 shift/reduce conflict.
*
* 2. attribute_name -> name Apostrophe attribute_designator .aggregate.
*causes 1 shift/reduce conflict (since .aggregate. -> | aggregate)
*
* 3. component_instantiation_statement -> a_label name
*.generic_map_aspect. .port_map_aspect. Semicolon_ERR
*causes 1 shift/reduce conflict (with .generic_map_aspect.).
*
*Reduce/reduce conflicts:
*
* 1. range -> attribute_name
*and name -> attribute_name causes 1 reduce/reduce conflict.
*
* 2. expanded_name -> simple_name
*and name -> simple_name  causes 2 reduce/reduce conflicts.
*
*To avoid conflicts while implementing on an LALR(1) shift-reduce
*parser-generator such as YACC or BISON, the original IEEE-1076 VHDL grammar
*has been modified at appropriate places:
*
*The production for formal_port_element contains "type_mark .constraint."
*instead of ".name. type_mark .constraint." (ie instead of subtype_indication)
*(AFIT file contains only "type_mark")
*
*The production for formal_generic_element contains "type_mark .constraint."
*instead of ".name. type_mark .constraint." (ie instead of subtype_indication)
*(AFIT file contains only "type_mark")
*
*In the production for architecture_body, "simple_name" (AFIT) has been
*changed to "name" in accordance with the LRM
*
*The production for configuration_declaration contains "name" instead of
*"entity_name" (LRM), or "Identifier" (AFIT)
*
*Missing Semicolon_ERR at end of production for block_configuration (in AFIT
```

```
*file) has been set right.
*
*In production for block_specification, "name" causes conflict and has not
*been implemented.
*
*In production for component_configuration, "Identifier" (AFIT) has been
*replaced by "name" in accordance with the LRM definition. Besides, missing
*Semicolon_ERR (in AFIT file) has been set right.
*
*In production for operator_symbol, "sign" has not been implemented. Besides,
*"StringLit" (absent in AFIT file) has been added in accordance with the LRM
*definition. Also, "logical_operator" and "miscelaneous_operator", and
*productions for them have been added (these were commented out in the AFIT
*file).
*
*In production for procedure_parameter_element, ".name. type_mark
*.constraint." (or subtype_indication) has been replaced with "type_mark
*.constraint.". AFIT file contains only "type_mark".
*
*In production for function_parameter_element, ".name. type_mark
*.constraint." (or subtype_indication) has been replaced with "type_mark
*.constraint.". AFIT file contains only "type_mark".
*
*In production for scalar_type_definition, "range_type_definition" includes
*both integer and floating point types.
*
*In production for index_subtype_definition, "type_mark" (LRM) has been
*replaced by "name".
*
*In production for discrete_range, "subtype_indication" (ie ".name.
*type_mark .constraint." in LRM) has been replaced by "name range_constraint
*| type_mark". Note: "constraint" (LRM) implies "range_constraint" or
*"index_constraint", but "index_constraint" has been ommitted in the
*production for discrete_range. This is the same as the AFIT file, except that
*"type_mark" has also been ommitted in AFIT file, since it causes 2
*reduce/reduce errors.
*
*Missing Semicolon_ERR in AFIT file for the production for
*incomplete_type_declaration has been set right here.
*
*This file as well as AFIT file contains "expanded_name" in production for
*"type_mark" to avoid conflict between "type_mark" and "constraint".
*
*In production for constraint, "index_constraint" has been replaced by
*"aggregate", both in this as well as the AFIT file.
*
*Missing Semicolon_ERR in AFIT file for the production for file_declaration
*has been set right.
*
*In production for association_element, ".formal_part_Arrow. actual_part"
*has been replaced by "name Arrow OPEN_or_expression | OPEN_or_expression".
```

```
*
*Productions for "formal_part" and "actual_part" have been replaced by their
*equivalents. (ie formal_part -> name | LeftParen name RightParen;
*actual_part -> OPEN_or_expression | LeftParen OPEN_or_expression RightParen;)
*
*In production for local_port_element, "subtype_indication .BUS.
*.VarAsgn_expression." has been replaced by "type_mark .constraint." in this
*file, and "type_mark" in the AFIT file.
*
*In production for local_generic_element, "subtype_indication
*.VarAsgn_expression." has been replaced by "type_mark .constraint." in this
*file, and "type_mark" in the AFIT file.
*
*In production for configuration_specification, "Identifier" (AFIT) has been
*replaced by "name" in this file in accordance with the LRM definition.
*
*In production for entity_aspect, "ENTITY Identifier" (AFIT) has been
*replaced by "ENTITY name" as per the LRM, but "CONFIGURATION name" (LRM)
*has been replaced by "CONFIGURATION Identifier", here, as well as in AFIT file.
*
*Missing Semicolon_ERR in production for disconnection_specification in AFIT
*file has been set right.
*
*In production for name, "indexed_name" includes "slice_name". Besides,
*name -> operator_symbol (operator overloading) has not been implemented.
*(causes 28 reduce/reduce conflicts).
*
*prefix -> function_call is not implemented. "function_call" is handled by
*"indexed_name".
*
*suffix -> operator_symbol is not implemented.
*
*indexed_name -> prefix ( expression ,{ expression } ) in LRM is implemented
*here as indexed_name -> name aggregate.
*
*In production for attribute_name, "prefix" (LRM) is replaced by "name",
*and optional '(' expression ')' in LRM is implemented as ".aggregate."
*here.
*
*"attribute_designator -> simple_name | RANGE" includes the keyword "RANGE"
*here. (used as an Identifier here).
*
*In production for primary, "function_call" is handled by "name", and '('
*expression ')' is handled by aggregate. Besides primary -> type_conversion
*is not implemented.
*
*literal -> Identifier is not implemented. (causes 99 reduce/reduce conflicts).
*
*Production for element_association contains "simple_expression direction
*simple_expression | name range_constraint" to compensate for change in
*production for "choice".
```

```
*
*choice -> discrete_range has been replaced by "choice -> simple_expression
*direction simple_expression | name range_constraint", since "discrete_range
*-> subtype_indication | range" causes conflicts. Besides "choice ->
*simple_expression | simple_name" has been replaced by "choice ->
*simple_expression" since "simple_expression" contains "simple_name" in LRM
*definition.
*
*function_call is handled by "indexed_name"
*
*In production for qualified_expression, "type_mark" has been replaced by
*"name", and "aggregate" includes '(' expression ')'.
*
*"type_conversion" has been replaced everywhere by its appropriate
*production.
*
*allocator -> NEW subtype_indication | NEW qualified_expression has been
*replaced by "NEW qualified_expression" only, since "subtype_indication"
*causes conflicts.
*
*.AFTER__expression. -> | AFTER numeric_literal (AFIT) has been changed to
*".AFTER__expression. -> | AFTER expression" to reflect the LRM.
*
*In production for procedure_call_statement, "actual_parameter_part" has
*been ommitted. Its inclusion causes 1 shift/reduce, and 2 reduce/reduce
*conflicts. Here, procedure_call_statement has been implemented as "name
*Semicolon_ERR", since "name" includes "name aggregate".
*
*In production for component_instantiation_statement, "Identifier" (AFIT)
*has been replaced by "name" as per the LRM definition
*
*generate_statement is always labelled (LRM). So unlabelled_generate_statement
*(AFIT) is not implemented.
*
*Missing Semicolon_ERR in production for library_clause in AFIT file has
*been set right.
*
*
*****************************************************************************/
/*
** $Header: vhdl.y,v 4.0 87/11/30 15:58:01 rbratton Exp $<y_op>$<y_op>$
**
** $Log: vhdl.y,v $<y_op>$<y_op>$
 * Revision 4.0  87/11/30  15:58:01  rbratton
 * Check in of VHDL version 4.0 (version reported in thesis).
 *
 * Revision 3.2  87/11/04  16:10:48  rbratton
 * Parser:  corrected ranges and aggregate grammar.  1 shift/reduce
 * conflict.
 * Lex:  Save before trying to implement alternate replacement
 * characters (! for |, : for #, and % for ").
```

```
 *
 * Revision 3.1  87/11/01  11:28:31  rbratton
 * Checkpoint save before trying to resolve "range" problems.
 *
 * Revision 3.0  87/10/15  06:23:49  rbratton
 * Beta 3 Save.  Implemented case/selected signal assignment and
 * with/use (using improved symbol table).
 *
 * Revision 2.3  87/10/11  15:06:54  rbratton
 * Because of problems with passing floating point parameters, floating
 * point has been removed--replaced with integer long.  Hopefully, at a
 * later time, the problems will be resolved.
 * This is also a configuration save before adding WITH/USE capabilities
 * to the analyzer.
 *
 * Revision 2.2  87/09/06  20:05:55  rbratton
 * Checkpoint save before implementing improved symbol table.
 *
 * Revision 2.1  87/09/01  11:26:46  rbratton
 * Implemented floating point notation.  Uses float (32 bits?) rather than
 * double, but could possibly be changed later.
 *
** Revision 2.0  87/08/29  09:43:08  rbratton
** Configuration save.  For VHDL Release 2.0
**
** Revision 1.8  87/08/24  18:30:11  rbratton
** 1 shift/reduce conflict (default acceptable).  Creates 487 cases.
** Changed value of NULL_SYMBOL from (struct sym_entry *) 0 to
** NULL (= 0).  Still creates a NULL pointer, but does not generate
** warnings while compiling the resulting code (vhdlyacc.c).
**
** Revision 1.7  87/08/18  19:35:46  rbratton
** Corrected problems with signal assignment statement.  Added labels to
** block statement and label symbol table entry.
**
** Revision 1.6  87/08/09  19:34:47  rbratton
** This version will NOT compile.  It causes a "switch table overflow".
** The next version may be a reduced grammar to try to avoid this
** problem.
**
** Revision 1.5  87/07/18  19:14:53  rbratton
** checkpoint save:  no conflicts
**
** Revision 1.4  87/07/17  18:21:23  rbratton
** checkpoint save:  9 shift/reduce conflicts
** Plus/Minus LeftParen
**
** Revision 1.3  87/07/17  17:57:50  rbratton
** checkpoint save:  13 shift/reduce conflicts
** Plus/Minus; Identifier
**
```

```
** Revision 1.2  87/07/15  10:07:55  rbratton
** checkpoint save
**
** Revision 1.1  87/06/21  09:24:24  rbratton
** Added some error recovery.  More to follow.
**
** Revision 1.0  87/04/24  17:28:14  rbratton
** Initial revision
**
**
*/


%{


#include <malloc.h> /* !!!!!! For BISON CODE !!!!!!*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "comp.h"
#include "arch.h"
#include "misc.h"
#include "signal.h"
#include "process.h"
#include "ident.h"
#include "comp_in.h"
#include "portmap.h"
#include "assoc.h"
#include "thesis.h"
#include "mcode.h"
#include "entity.h"
#include "port.h"
#include "generate.h"

#include "vhdl.hpp"

int G_translate = TRUE;  // Translate signal IDs to offsets if TRUE

extern int ANY_NAME; /* generic hash table index for error recovery */
int op1, op2, op3;      /* temporary variables for op indices */

int is_childless; /* attribute of architecture body */
int is_structure; /* attribute of architecture body */
%}

%union  {
int y_tok; /*   token */
int y_op; /* Index to op table entry    */
int y_hash; /* Index to hash table entry  */
int y_str;     /* Index to string storage  */
```

```
/*SYM_PTR y_sym;*/ /* Pointer to symbol table entry   */
long   y_val;  /* Floating point number (32 bits) */
/*   (also handles integer values) */
}

/* terminal symbols */

/* old terminal symbols - keep until removed from yacc code */


/*%token '&' */
%token Apostrophe
/*%token '('*/
/*%token RightParen*/
%token DoubleStar
/*%token Star */
/*%token '+' */
/*%token ',' */
/*%token '-'  */
%token VarAsgn
/*%token ':' */
/*%token Semicolon */
%token LESym
%token Box
/*%token '<'  */
%token Arrow
/*%token '=' */
%token GESym
/*%token '>' */
%token Bar
%token NESym
/*%token '.' */
%token Slash
%token Identifier
%token DecimalInt
%token DecimalReal
%token BasedInt
%token BasedReal
%token CharacterLit
%token StringLit
%token BitStringLit
%token  ABS
%token  ACCESS
%token  AFTER
%token  ALIAS
%token  ALL
%token  AND
%token  ARCHITECTURE
%token  ARRAY
%token  ASSERT
%token  ATTRIBUTE
```

```
%token  BEGIN_
%token  BIT
%token  BLOCK
%token  BODY
%token  BUFFER
%token  BUS
%token  CASE
%token  COMPONENT
%token  CONFIGURATION
%token  CONSTANT
%token  DISCONNECT
%token  DOWNTO
%token  ELSE
%token  ELSIF
%token  END_
%token  ENTITY
%token  EXIT
%token  FILE_
%token  FOR
%token  FUNCTION
%token  GENERATE
%token  GENERIC
%token  GUARDED
%token  IF
%token  INOUT
%token  IN
%token  IS
%token  LABEL
%token  LIBRARY
%token  LINKAGE
%token  LOOP
%token  MAP
%token  MOD
%token  NAND
%token  NEW
%token  NEXT
%token  NOR
%token  NOT
%token  NULL_
%token  OF
%token  ON
%token  OPEN
%token  OR
%token  OTHERS
%token  OUT
%token  PACKAGE
%token  PORT
%token  PROCEDURE
%token  PROCESS
%token  RANGE
%token  RECORD
```

```
%token  REGISTER
%token  REM
%token  REPORT
%token  RETURN
%token  SELECT
%token  SEVERITY
%token  SIGNAL
%token  SUBTYPE
%token  THEN
%token  TO
%token  TRANSPORT
%token  TYPE
%token  UNITS
%token  UNTIL
%token  USE
%token  VARIABLE
%token  WAIT
%token  WHEN
%token  WHILE
%token  WITH
%token  XOR

 /*****************************************************************************
 **
 **  operator precedences and associativities listed in
 **  increasing precedence_
 **
 **  Note:  ABS and NOT have the same precedence as DoubleStar;
 **  yet, they associate to the right_  The (non)token UNARY_SIGN is used
 **  only to establish precedence for unary plus/minus signs_  It does not
 **  have to be a declared token or have any other value other than its
 **  relative precedence value_
 **
 *****************************************************************************/

%left AND OR NAND NOR XOR
%left '=' NESym '<' LESym '>' GESym
%left '+' '-' '&'
%left '*' Slash MOD REM
%right UNARY_SIGN
%left DoubleStar
%right ABS NOT

%{
#ifndef NDEBUG
#define TRACE(x,z) {if(yaccdebug)printf("#RULE %s ::= %s\n",x,z);}
#else
#define TRACE(x,z) ;
#endif
%}
```

```
/*
 *  Start symbol = "design_file"
 */
%start design_file

/*
 *  Rules
 */
%%

/*
**  Chapter 1: Design Entities
*/
/* 1_1 */
entity_declaration
: ENTITY
      {
      }
  Identifier
      {
         entity_add();
      }
  IS
  _generic_clause_
     {
        port_clear();
     }
  _port_clause_
     {
        entity_add_port();
     }
  entity_declarative_part
  _BEGIN__entity_statement_part_
  END_ERR
  _simple_name_
  Semicolon_ERR
| ENTITY error
;

/* 1_1_1 */
_port_clause_
: /*empty*/
| port_clause
;


_generic_clause_
: /*empty*/
| generic_clause
;
```

```
port_clause
: PORT
  '('
  formal_port_list
  RightParen_ERR
  Semicolon_ERR
;

generic_clause
: GENERIC
  '('
  formal_generic_list
  RightParen_ERR
  Semicolon_ERR
;



/* 1_1_1_2 */
formal_port_list
: formal_port_element
  ___formal_port_element__
| error RightParen_ERR
;


___formal_port_element__
: /*empty*/
| ___formal_port_element__
  Semicolon_ERR
  formal_port_element
{
yyerrok;
}
;

formal_port_element
: _SIGNAL_
      {
         ident_list_clear();
      }
  identifier_list
  ':'
  _mode_
  /* _name_ causes conflict  */
  type_mark
  _constraint_
  _BUS_
      {
         port_add_id_list();
      }
```

```
  _VarAsgn__expression_
;

_SIGNAL_
: /*empty*/
| SIGNAL
;

_mode_
:/*empty*/
| IN
      {
         direct_set(V_IN);
      }
| OUT
      {
         direct_set(V_OUT);
      }
| INOUT
| BUFFER
| LINKAGE
;

_BUS_
: /*empty*/
| BUS
;



_VarAsgn__expression_
: /*empty*/
| VarAsgn
  expression
;

/* 1_1_1_1 */
formal_generic_list
: formal_generic_element
  ___formal_generic_element__
| error RightParen_ERR
;


___formal_generic_element__
: /*empty*/
| ___formal_generic_element__
  Semicolon_ERR
  formal_generic_element
   {
   yyerrok;
```

```
}
;

formal_generic_element
: _CONSTANT_
  identifier_list
  ':'
  _IN_
  /* _name_ causes conflict */
  type_mark
  _constraint_
  _VarAsgn__expression_
;


_CONSTANT_
: /*empty*/
| CONSTANT
;

_IN_
: /*empty*/
| IN
;

/* 1_1_2 */
entity_declarative_part
: __entity_declarative_item__
;


__entity_declarative_item__
: /*empty*/
| __entity_declarative_item__
  entity_declarative_item
;


entity_declarative_item
: alias_declaration
| constant_declaration
| type_declaration
| subtype_declaration
| attribute_declaration
| attribute_specification
| subprogram_declaration
| subprogram_body
| signal_declaration
| file_declaration
| disconnection_specification
| use_clause
```

```
;


/* 1_1_3 */

_BEGIN__entity_statement_part_
: /* empty */
|  BEGIN_
  entity_statement_part
;


entity_statement_part
: __entity_statement__
;

__entity_statement__
: /*empty*/
| __entity_statement__
  entity_statement
;


entity_statement
: concurrent_assertion_statement
| concurrent_procedure_call
| process_statement  /* NOT IN 7_2 */
;


/* 1_2 */
/* architecture bodies */

architecture_body
: ARCHITECTURE
      {
      }
  Identifier
     {
        arch_add();
     }
  OF
  name  /* entity name */
     {
        arch_name();
     }
  IS
     {
        signal_clear_list();
        comp_clear_list();
        comp_inst_list_clear();
```

```
    }
  architecture_declarative_part
  BEGIN_
  architecture_statement_part
  END_ERR
  _simple_name_   /* architecture simple name */
  Semicolon_ERR
| ARCHITECTURE error
;

/* 1_2_1 */
/* Architecture Declarative Part */
architecture_declarative_part
: __block_declarative_item__
;

__block_declarative_item__
: /*empty*/
| __block_declarative_item__
  block_declarative_item
;

block_declarative_item
: constant_declaration
| signal_declaration
    {
       arch_add_signal_list();
    }
| type_declaration
| subtype_declaration
| attribute_declaration
| component_declaration
    {
       arch_add_comp_list();
    }
| alias_declaration
| attribute_specification
| configuration_specification
   | subprogram_declaration
| subprogram_body
| file_declaration
| disconnection_specification
| use_clause
;

/* 1_2_2 */
/* Architecture Statement Part */
architecture_statement_part
: __concurrent_statement__
;
```

```
/* 1_3  */
configuration_declaration
: CONFIGURATION
     {
     }
  Identifier
     {
        generate_got_top_id(ident_get())
     }
  OF
  name  /* Identifier */   /* entity_name */
     {
        generate_got_top_entity_id(ident_get())
     }
  IS
  configuration_declarative_part
  block_configuration
  END_ERR
  _simple_name_
  Semicolon_ERR
| CONFIGURATION error
;

configuration_declarative_part
: __configuration_declarative_item__
;

__configuration_declarative_item__
: /*empty*/
| __configuration_declarative_item__
  configuration_declarative_item
;

configuration_declarative_item
: use_clause
| attribute_specification
;

/* 1_3_1 */
/* block configuration */
block_configuration
: FOR
  block_specification
     {
        generate_got_top_arch_id(ident_get());
     }
  __use_clause__
  __configuration_item__
  END_ERR
  FOR
```

```
  Semicolon_ERR
;


__use_clause__
: /*empty*/
| use_clause
  __use_clause__
;

__configuration_item__
: /*empty*/
| __configuration_item__
  configuration_item
;

block_specification
: label /* arch, block, generate */
  _opt_index_spec_
/*  | name causes conflict */
;

_opt_index_spec_
: /*empty*/
| '('
  index_specification
  RightParen_ERR
;


index_specification
: discrete_range
| expression
;

configuration_item
: block_configuration
| component_configuration
;

/* 1_3_2 */

component_configuration
: FOR
  instantiation_list
  ':'
  name  /* Identifier  */
        _USE__binding_indication_
  _block_configuration_
  END_ERR
  FOR
```

```
    {
       generate_ident_list();
    }
  Semicolon_ERR
;

_USE__binding_indication_
: /*empty*/
| USE
  binding_indication
  Semicolon_ERR
;

_block_configuration_
: /*empty*/
| block_configuration
;


/*
**  Chapter 2:  Subprograms
*/

/* 2_1 */
subprogram_declaration
: subprogram_specification
  Semicolon_ERR
;

subprogram_specification
: PROCEDURE
  designator
  _procedure_parameter_list_
| FUNCTION
  designator
  _function_parameter_list_
  RETURN
  type_mark
;

designator
: Identifier
| operator_symbol
;

operator_symbol    /* defined in LRM 2_1 */
: relational_operator
| adding_operator
/* | sign  */
| multiplying_operator
| logical_operator
```

```
| miscellaneous_operator
| StringLit
;
logical_operator : AND | OR | NAND | NOR | XOR
;
miscellaneous_operator : DoubleStar | ABS | NOT
;

_procedure_parameter_list_
: /*empty*/
| '('
  procedure_parameter_element
  ___procedure_parameter_element__
  RightParen_ERR
| '('
  error
  RightParen_ERR
;


___procedure_parameter_element__
: /*empty*/
| ___procedure_parameter_element__
  Semicolon_ERR
  procedure_parameter_element
{
yyerrok;
}
;


procedure_parameter_element
: _procedure_parameter_object_class_
  identifier_list
  ':'
  _procedure_parameter_mode_
  /* _name_  causes conflict */
  type_mark
  _constraint_
  _VarAsgn__expression_
;

_procedure_parameter_object_class_
: /*empty*/
| VARIABLE
| CONSTANT
;

_procedure_parameter_mode_
: /*empty*/
| IN
```

```
| OUT
| INOUT
;

_function_parameter_list_
: /*empty*/
| '('
  function_parameter_element
  ___function_parameter_element__
  RightParen_ERR
| '('
  error
  RightParen_ERR
;


___function_parameter_element__
: /*empty*/
| ___function_parameter_element__
  Semicolon_ERR
  function_parameter_element
;

function_parameter_element
: _function_parameter_object_class_
  identifier_list
  ':'
  _function_parameter_mode_
  type_mark
  _constraint_
  _VarAsgn__expression_
;


_function_parameter_object_class_
: /*empty*/
| CONSTANT
| SIGNAL
;

_function_parameter_mode_
: /*empty*/
| IN
;


/* 2_2 */
subprogram_declarative_part
: /*empty*/
| subprogram_declarative_part
  subprogram_declarative_item
```

```
{
yyerrok;
}
;

subprogram_declarative_item
: constant_declaration
| variable_declaration
| alias_declaration
| type_declaration
| subtype_declaration
| attribute_declaration
| attribute_specification
| subprogram_declaration
| subprogram_body
| file_declaration
| use_clause
;


/* 2_2 */
/* subprogram bodies */

subprogram_body
: subprogram_specification
  IS
  subprogram_declarative_part
  BEGIN_
  sequence_of_statements
  END_ERR
  _designator_
  Semicolon_ERR
;

_designator_
: /*empty*/
| designator
;



/* Packages */

/* 2_5 */
package_declaration
: PACKAGE
  Identifier
  IS
  package_declarative_part
  END_ERR
  _simple_name_
```

```
  Semicolon_ERR
| PACKAGE
error
;

package_declarative_part
: __package_declarative_item__
;

__package_declarative_item__
: /*empty*/
| __package_declarative_item__
  package_declarative_item
;

package_declarative_item
: type_declaration
| subtype_declaration
| attribute_declaration
| constant_declaration
| alias_declaration
| subprogram_declaration
| component_declaration
| attribute_specification
| signal_declaration
| file_declaration
| disconnection_specification
| use_clause
| error END_ERR
  Semicolon_ERR
;

/* 2_6 */
/* package bodies */

package_body
: PACKAGE
  BODY
  Identifier
  IS
  package_body_declarative_part
  END_ERR
  _simple_name_
  Semicolon_ERR
| PACKAGE
     BODY
     error
;

package_body_declarative_part
: __package_body_declarative_item__
```

```
;

__package_body_declarative_item__
: /*empty*/
| __package_body_declarative_item__
  package_body_declarative_item
;

package_body_declarative_item
: subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| file_declaration
| alias_declaration
| use_clause
;


/*
**  Chapter 3:  Types
*/


/* 3_1 */
scalar_type_definition
: enumeration_type_definition
| range_type_definition  /* includes integer and floating point */
| physical_type_definition
;

range_constraint
: RANGE
  range
;

range
: attribute_name /* simple_expression simple_expression -> (attribute) name */
| simple_expression
  direction
  simple_expression
;

direction
: TO
| DOWNTO
;

/* 3_1_1_ */
enumeration_type_definition
```

```
: '('
  enumeration_literal
  ___enumeration_literal__
  RightParen_ERR
;


___enumeration_literal__
: /*empty*/
| ___enumeration_literal__
  ','
  enumeration_literal
{
yyerrok;
}
;

enumeration_literal
: Identifier
| CharacterLit
;

/* 3_1_2 & 3_1_4 */
/* Integer and Floating Point types */

range_type_definition
: range_constraint
;

/* 3_1_3 */
physical_type_definition
: range_constraint
  UNITS
  base_unit_declaration
  __secondary_unit_declaration__
  END_ERR
  UNITS
;


__secondary_unit_declaration__
: /*empty*/
| __secondary_unit_declaration__
  secondary_unit_declaration
{
yyerrok;
}
;

base_unit_declaration
: Identifier
```

```
  Semicolon_ERR
;

secondary_unit_declaration
: Identifier
  '='
  physical_literal
  Semicolon_ERR
;

physical_literal
: _abstract_literal_
  name      /* in LRM: UNIT_name */
;


_abstract_literal_
: /*empty*/
| abstract_literal
;

/* 3_2 */
composite_type_definition
: array_type_definition
| record_type_definition
;

/* 3_2_1 */
array_type_definition
: unconstrained_array_definition
| constrained_array_definition
;

unconstrained_array_definition
: ARRAY
  '('
  index_subtype_definition
  ___index_subtype_definition__
  RightParen_ERR
  OF
  subtype_indication
;


___index_subtype_definition__
: /*empty*/
| ___index_subtype_definition__
  ','
  index_subtype_definition
;
```

```
constrained_array_definition
: ARRAY
  index_constraint
  OF
  subtype_indication
;

index_subtype_definition
: name    /* type_mark  - causes conflict */
  RANGE
  Box
;

index_constraint
: '('
  discrete_range
  ___discrete_range__
  RightParen_ERR
;

___discrete_range__
: /*empty*/
| ___discrete_range__
  ','
  discrete_range
{
yyerrok;
}
;

discrete_range
: range  /* includes attribute_name */
| name
  range_constraint   /* subtype_indication - causes conflict */
| type_mark  /* type_mark causes 2 r/r conflicts - required for
   louie's code */
;

/* 3_2_2 */
record_type_definition
: RECORD
  element_declaration
  __element_declaration__
  END_ERR
  RECORD
| RECORD
  error
  END_ERR
  RECORD
;
```

```
__element_declaration__
: /*empty*/
| __element_declaration__
  element_declaration
  {
  yyerrok;
  }
;

element_declaration
: identifier_list
  ':'
  element_subtype_definition
  Semicolon_ERR
;

/*
**  identifier_list is used consistantly in definitions of new Identifiers,
**  with one exception--IMPORT_DIRECTIVE_  The IMPORT_DIRECTIVE expects to
**  find all Identifiers declared at the local scope and it is an error if
**  they are not_  In all other cases, it is an error to have two Identifiers
**  with the same name at the same level_  (Overloading not implemented_)
**  Therefore, identifier_list checks to see if the previous token was
**  IMPORT_
**  returns pointer to symbol table which has a list of identifier
**  definitions connected by the "next" pointers_
*/
identifier_list
: Identifier
      {
         ident_list_clear();
         ident_list_add();
         #ifdef _DB1_
         puts("~~~ Enter_id_list()");
         #endif
      }
  ___identifier__
;


___identifier__
: /*empty*/
| ','
  Identifier
     {
        ident_list_add();
        #ifdef _DB1_
        puts("~~~ Enter_id_list()");
        #endif
     }
```

```
  ___identifier__
;


element_subtype_definition
: subtype_indication
;

/* 3_3 */
/* Access Types */
access_type_definition
: ACCESS
  subtype_indication
;

/* 3_3_1 */
/* Incomplete Type Declarations */
incomplete_type_declaration
: TYPE
  Identifier
  Semicolon_ERR
;


/* 3_4 */
/* File Types */
file_type_definition
: FILE_
  OF
  type_mark
;



/*
**  Chapter 4:  Declarations
*/
/* 4_1_1 */
type_declaration
: full_type_declaration
| incomplete_type_declaration
;

full_type_declaration
: TYPE
  Identifier
  IS
  type_definition
  Semicolon_ERR
;
```

```
type_definition
: scalar_type_definition
| composite_type_definition
| access_type_definition
| file_type_definition
| error Semicolon_ERR
;

/* 4_2 */
subtype_declaration
: SUBTYPE
  Identifier
  IS
  subtype_indication
    Semicolon_ERR
;

subtype_indication
: type_or_function_name
  type_mark
  _constraint_
| type_mark
  _constraint_
;


_constraint_
: /*empty*/
| constraint
;

type_or_function_name
: expanded_name
;


expanded_name
: simple_name  /* Identifier  */
/* | STANDARD
 */
| expanded_name  /* was Identifier  */
  '.'
  simple_name  /*  Identifier */
   {
yyerrok;
   }
;

/*!!!!!!!!!!!! CHANGED !!!!!!!!!!!!!!!!
!type_mark
! : expanded_name / * move to production_c !!! * /
```

```
!
! / * type_mark and constraint
!    will otherwise cause conflict * /
! ;
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
type_mark
: BIT
      {
         type_set(V_BIT);
      } /* Only allow BIT types at this time */
;

/*!!!!!!!!!!!!!! END CHANGE !!!!!!!!!!!!!!!*/

constraint
: range_constraint
| aggregate   /* was: ( discrete_range ___discrete_range__ ) */
         /* index_constraint */
;

/* 4_3_1_1  */
constant_declaration
: CONSTANT
  identifier_list
  ':'
  subtype_indication
  _VarAsgn__expression_
  Semicolon_ERR
;

/* 4_3_1_2 */
signal_declaration
: SIGNAL
      {
         ident_list_clear();
      }
  identifier_list
  ':'
  subtype_indication
  _signal_kind_
     {
        signal_add_id_list();
     }
  _VarAsgn__expression_
  Semicolon_ERR
;

_signal_kind_
: /*empty*/
| signal_kind
;
```

```
signal_kind
: REGISTER
| BUS
;

/* 4_3_1_3 */
variable_declaration
: VARIABLE
  identifier_list
  ':'
  subtype_indication
  _VarAsgn__expression_
  Semicolon_ERR
;

/* 4_3_2 */
/* File Declarations */
file_declaration
: FILE_
  Identifier
  ':'
  subtype_indication
  IS
  _mode_
  expression
  Semicolon_ERR
;

/* 4_3_3 and 4_3_3_1 */
/* Interface Declaration and lists are interspersed
   where they are actually used port, generic and parameter */

/* 4_3_3_2 */
/* Association lists */

association_list
: association_element
  ___association_element__
;

___association_element__
: /*empty*/
| ','
  association_element
  ___association_element__
   {
yyerrok;
}
;
```

```
/*
**  ( expression ) is defined by aggregate as:
** ( general_element_association ) =>
** ( OPEN_or_expression ) =>
** ( expression )
*/

association_element
: /* formal_part */  /* causes conflict */
  name
      {
         assoc_list_add_node();
         assoc_left(ident_get());
      }
  Arrow
      {
         mcode_clear_list();
         G_translate = FALSE; // Stop translating signals to offsets
      }
  /* actual_part */
  OPEN_or_expression   /* can be name also */
   {
         int signal_name;
         mcode_c_pop_top();
         signal_name = mcode_c_pop();
         assoc_right(signal_name);
         G_translate = TRUE; // Start translating signals to offsets
      }
| /* actual_part */
OPEN_or_expression   /* can be name also */
;
/* causes conflict
formal_part : name | name '(' name  ')'
;
actual_part : OPEN_or_expression | name '(' OPEN_or_expression
')'
;
*/
OPEN_or_expression
: OPEN
| expression
;

/* 4_3_4 */
alias_declaration
:  ALIAS
     Identifier
':'
subtype_indication
IS
name
```

```
Semicolon_ERR
;


/* 4_4 */
attribute_declaration
: ATTRIBUTE
  Identifier
  ':'
  type_mark
  Semicolon_ERR
;


/* 4_5 */
component_declaration
: COMPONENT
  Identifier
     {
        comp_add_comp();
     }
  _GENERIC__local_generic_list_
  _PORT__local_port_list_
     {
        comp_add_port();
     }
  END_ERR
  COMPONENT
  Semicolon_ERR
;

_PORT__local_port_list_
: /*empty*/
| PORT
      {
         port_clear();
      }
  '('
  local_port_list
  RightParen_ERR
  Semicolon_ERR
;

local_port_list
: local_port_element
  ___local_port_element__
| error RightParen_ERR
;


___local_port_element__
```

```
: /*empty*/
| ___local_port_element__
  Semicolon_ERR
  local_port_element
;


local_port_element
: _SIGNAL_
      {
         ident_list_clear();
      }
  identifier_list
  ':'
  _local_port_mode_
  type_mark
     {
        port_add_id_list();
     }
  _constraint_
;

_local_port_mode_
: /*empty*/
| IN
      {
         direct_set(V_IN);
      }
| OUT
      {
         direct_set(V_OUT);
      }
| INOUT
| BUFFER
| LINKAGE
;

_GENERIC__local_generic_list_
: /*empty*/
| GENERIC
  '('
  local_generic_list
  RightParen_ERR
  Semicolon_ERR
;

local_generic_list
: local_generic_element
  ___local_generic_element__
| error RightParen_ERR
;
```

```
___local_generic_element__
: /*empty*/
| ___local_generic_element__
  Semicolon_ERR
  local_generic_element
;


local_generic_element
: _CONSTANT_
  identifier_list
  ':'
  _IN_
  type_mark
  _constraint_
;

/*
**  Chapter 5:  Specifications
*/

/* 5_1 */
attribute_specification
: ATTRIBUTE
  attribute_designator
  OF
  entity_specification
  IS
  expression
  Semicolon_ERR
;

entity_specification
: entity_name_list
  ':'
  entity_class
;


entity_class
: ENTITY
| ARCHITECTURE
| PACKAGE
| FUNCTION
| PROCEDURE
| SUBTYPE
| CONSTANT
| VARIABLE
| SIGNAL
```

```
| LABEL
| TYPE
| CONFIGURATION
| COMPONENT
;

entity_name_list
: entity_designator
  ___entity_designator__
| OTHERS
| ALL
;

___entity_designator__
: /*empty*/
| ___entity_designator__
  ','
  entity_designator
;

entity_designator
: simple_name
| operator_symbol
;


/* 5_2 */
configuration_specification
: FOR
  instantiation_list
     {
        ident_c_list_print();
     }
  ':'
  /* Identifier */
  name
  USE
  binding_indication
  Semicolon_ERR
| FOR
  error
  Semicolon_ERR
;

instantiation_list
: identifier_list
| OTHERS
| ALL
| error ':'
{
yyerrok;
```

```
}
;

/* 5_2_1 */
binding_indication
: entity_aspect
  _generic_map_aspect_
  _port_map_aspect_
;


/* 5_2_1_1 */
entity_aspect
: ENTITY
  /* Identifier */
/*   name  / * name causes 1 s/r conflict */
     Identifier
     {
        generate_entity_name(ident_get());
     }
  _architecture_identifier_
     {
        generate_arch_name(ident_get());
     }
| CONFIGURATION
  Identifier
  /* name  causes conflict */
     {
        printf("!!!! NOT IMPLEMENTED !!!!!!! %d\n", ident_get());
     }
| OPEN
;

_architecture_identifier_
: /*empty*/
| '('
  simple_name
  RightParen_ERR
;


/* entity_indication
: library_name   entity library_name
| OPEN
; */

/* 5_2_1_2 */
___element_association__
: /*empty*/
| ___element_association__
  ','
```

```
  element_association
  {
  yyerrok;
  }
;
/* 5_3 */
/* Disconnection_specification */
disconnection_specification
: DISCONNECT
  guarded_signal_specification
  AFTER
  expression
  Semicolon_ERR
;

guarded_signal_specification
: signal_list
  ':'
  type_mark
;



/* 6_2_3 */
/* initialize_directive
: INITIALIZE
  type_mark
  TO
  expression
  __waveform_
  Semicolon_ERR
;  */

/*
**  Chapter 6:  Names
*/

/* 6_1 */
/*
** According to the VHDL Test suite, library names are not used
** in expressions_  Therefore, the choice "library_name" is removed_
**     NEED TO CHECK THIS OUT!!!  */
name
: simple_name   /* move to production_c */
| indexed_name     /* includes "slice_name" */
| selected_name
| attribute_name /*  not implemented: causes 2 reduce/reduce conflicts_
| operator_symbol   overloading not implemented
  causes reduce/reduce conflicts (28) */
;
```

```
prefix
: name  /*function call handled by indexed_name*/
/* | function_call
*/
;


/* 6_2 */
simple_name      /* returns hash index */
: Identifier
;


_simple_name_
: /*empty*/
| simple_name
;


/* 6_3 */
selected_name
: prefix
  '.'
  suffix
;

suffix
: simple_name
| CharacterLit
/* |  operator_symbol  */  /* handled by characterLit */
| ALL
;

/* 6_4 */
indexed_name    /* also includes "slice_name"   6_5 */
: name               /* in LRM: prefix */
  aggregate  /* in LRM:  '(' expression { ',' expression } ')'  */
;

/* 6_6 */
attribute_name
: name   /* prefix  causes 7 shift/reduce conflicts */
  Apostrophe
  attribute_designator
  _aggregate_  /* in LRM: '(' static_expression ')' */
;


_aggregate_
: /*empty*/
| aggregate
```

```
;

attribute_designator
: simple_name      /* attribute simple_name */
| RANGE      /*   somebody goofed!  Keyword used as an identifier */
;

/********
**
** Chapter 7  Expressions
**
*********/

/* 7_1 */
expression
: relation__AND__relation__
| relation__OR__relation__
| relation_NAND_NOR__relation_
| relation__XOR__relation__
;


relation__AND__relation__
: relation
  AND
  relation
      {
         mcode_add(M_AND);
      }
| relation__AND__relation__
  AND
  relation
      {
         mcode_add(M_AND);
      }
;

relation__OR__relation__
: relation
  OR
  relation
      {
         mcode_add(M_OR);
      }
| relation__OR__relation__
  OR
  relation
      {
         mcode_add(M_OR);
      }
;
```

```
relation_NAND_NOR__relation_
: relation
| relation
  NAND
  relation
      {
         mcode_add(M_NAND);
      }
| relation
  NOR
  relation
      {
         mcode_add(M_NOR);
      }
;


relation__XOR__relation__
: relation
  XOR
  relation
      {
         mcode_add(M_XOR);
      }
| relation__XOR__relation__
  XOR
  relation
      {
         mcode_add(M_XOR);
      }
;

relation
: simple_expression
  _relop__simple_expression_
;


_relop__simple_expression_
: /*empty*/
| relational_operator
  simple_expression
;

/*
** simple_expression ::= [sign] term { adding_operator term }
*/
simple_expression
: _sign_term__add_op__term__
;
```

```
relation_NAND_NOR__relation_
: relation
| relation
  NAND
  relation
      {
         mcode_add(M_NAND);
      }
| relation
  NOR
  relation
      {
         mcode_add(M_NOR);
      }
;


relation__XOR__relation__
: relation
  XOR
  relation
      {
         mcode_add(M_XOR);
      }
| relation__XOR__relation__
  XOR
  relation
      {
         mcode_add(M_XOR);
      }
;

relation
: simple_expression
  _relop__simple_expression_
;


_relop__simple_expression_
: /*empty*/
| relational_operator
  simple_expression
;

/*
** simple_expression ::= [sign] term { adding_operator term }
*/
simple_expression
: _sign_term__add_op__term__
;
```

```
term
: factor
| term
  multiplying_operator
  factor
{
yyerrok;
}
;


_sign_term__add_op__term__
: term   %prec UNARY_SIGN
| sign
  term   %prec UNARY_SIGN
| _sign_term__add_op__term__
  adding_operator
  term
;

factor
: primary
  _DoubleStar__primary_
| ABS
  primary
| NOT
  primary
     {
        // puts("~~~! mcode_add(NOT)");
        mcode_add(M_NOT);
     }
;


_DoubleStar__primary_
: /*empty*/
| DoubleStar
  primary
;

    primary
    : literal
         {
            mcode_add(lit_get());
         }
    | qualified_expression
/*      | function_call
*/
    | name     /* name = simple_name = Identifier = enumeration_literal */
       {
```

```
            if(G_translate == TRUE ) {
               TP_arch arch_ptr = arch_get(arch_c_get_id());
               TP_entity ent_ptr  = entity_get(arch_ptr->name);
               TP_port   port_ptr = ent_ptr->port_list;
               TP_port   the_port = port_get(ident_get(), port_ptr);

               mcode_add(the_port->number); // Offset of the port
            }
            else {
               mcode_add(ident_get());
            }
            mcode_add(M_GET);
         }
    /* includes function_call */
    | aggregate  /*(expression) is included under aggregate*/
    /* | type_conversion causes reduce/reduce conflicts
    */ | allocator
    ;

    /* 7_2_1 */
    /* logical operators embedded in expression */

    /* 7_2_2 */
    relational_operator
    : '='
    | NESym
    | '<'
    | LESym
    | '>'
    | GESym
    ;

    /* 7_2_3 */
    adding_operator
    : '+'
    | '-'
| '&'
;

sign
: '+'
| '-'
;

/* 7_2_4 */
multiplying_operator
: '*'
| Slash
| MOD
| REM
```

```
;

/* 7_3_1 */
literal
: numeric_literal
| CharacterLit
/*
 * | enumeration_literal
 *      Causes 99 reduce/reduce conflicts with Id and CharLit_
 *      Covered under 'name'_
 */ /* Identifier causes conflict */
| StringLit
| BitStringLit
| NULL_
;

numeric_literal
: abstract_literal
| abstract_literal      /* physical_literal */
  name /* Identifier */            /* in LRM: UNIT_name */

  /* name in physical_literal causes conflict */
;

/* 7_3_2 */
aggregate
: '('
  element_association
  ___element_association__
  RightParen_ERR
;

element_association
: expression
| choice
  __Bar__choice__
  Arrow
  expression
| simple_expression
  direction /* because of production for "choice"
                           to avoid conflict */
  simple_expression
| name
  range_constraint
;

choices
: choice
  __Bar__choice__
{
yyerrok;
```

```
}
;


__Bar__choice__
: /*empty*/
| __Bar__choice__
  Bar
  choice
;

choice
: simple_expression  /* includes simple_name */
| simple_expression
  direction
  simple_expression /* because of production for "discrete_range"
   to avoid conflict */
| name
  range_constraint
| OTHERS
;


/* 7_3_3 */
/* function_call
: Identifier
actual_parameter_part
;

actual_parameter_part
:
 '('
  association_list
  RightParen_ERR
;  function_call handled by selected name */
/* _actual_parameter_part_
:
| '('
  association_list
  RightParen_ERR
; * function_call handled by selected name */

/* 7_3_4 */
qualified_expression
: name
  Apostrophe
  aggregate
  /*
  ** type_mark ' aggregate | type_mark ' ( expression )
  */
;
```

```
/* 7_3_5 */
/* type_conversion causes reduce/reduce conflict
: Identifier  type_mark
  '('  aggregate
  expression
  RightParen_ERR
;
*/

/* 7_3_6 */
allocator
: NEW
/*   subtype_indication

| NEW     causes numerous reduce/reduce conflicts */
  qualified_expression
;


/*
**  Chapter 8:   Sequential Statements
*/

/* 8_0 */
sequence_of_statements
: __sequential_statement__
| error END_ERR
{
yyerrok;
}
| error ELSIF
{
yyerrok;
}
| error ELSE
{
yyerrok;
}
| error WHEN
{
yyerrok;
}
;


__sequential_statement__
: /*empty*/
| sequential_statement
__sequential_statement__
;
```

```
sequential_statement
: assertion_statement
| signal_assignment_statement
| variable_assignment_statement
| if_statement
| case_statement
| loop_statement
| next_statement
| exit_statement
| return_statement
| null_statement
| procedure_call_statement
| wait_statement
;

/* 8_1 */
wait_statement
: WAIT
  _sensitivity_clause_
  _condition_clause_
  _timeout_clause_
  Semicolon_ERR
;

_sensitivity_clause_
: /*empty*/
| sensitivity_clause
;

_condition_clause_
: /*empty*/
| condition_clause
;

_timeout_clause_
:  /*empty*/
| timeout_clause
;

sensitivity_clause
: ON
  /* sensitivity_list */
  signal_list
;

condition_clause
: UNTIL
  expression
;
```

```
timeout_clause
: FOR
  expression
;


/*
**  returns SYM_REF op tree indexes
*/
signal_list
: name
  ___name__
| OTHERS
| ALL
;


___name__
: /*empty*/
| ___name__
  ','
  name
;

/* 8_2 */
assertion_statement
: ASSERT
  expression
  _REPORT__expression_
  _SEVERITY__expression_
  Semicolon_ERR
;


_REPORT__expression_
: /*empty*/
| REPORT
  expression
;

_SEVERITY__expression_
: /*empty*/
| SEVERITY
  expression
;

/* 8_3 */
signal_assignment_statement
:
     target
     {
```

```
        TP_arch arch_ptr   = arch_get(arch_c_get_id());
        TP_entity ent_ptr  = entity_get(arch_ptr->name);
        TP_port   port_ptr = ent_ptr->port_list;
        TP_port   the_port = port_get(ident_get(), port_ptr);

        mcode_add(the_port->number); // Offset of the port
     }
  LESym
  _TRANSPORT_
     {
        G_translate = TRUE; // Start translating signals to offsets
     }
  waveform
  Semicolon_ERR
     {
        mcode_add(M_POST);
     }
;

target
: name
| aggregate
;


waveform
: waveform_element
  ___waveform_element__
;


___waveform_element__
: /*empty*/
| ','
  waveform_element
  ___waveform_element__
;

/* 8_3_1 */
waveform_element
: expression   /* NULL can be arrived at through expression - literal */
  _AFTER__expression_
;


_AFTER__expression_
: /*empty*/
| AFTER
  expression /* numeric_literal */   /* in LRM: expression */
;
```

```
/* 8_4 */
variable_assignment_statement
: target
  VarAsgn
  expression
  Semicolon_ERR
;

/* 8_5 */
procedure_call_statement
: name  /* name includes "name ( association_list )" */
  Semicolon_ERR      /* need to include actual_parameter_part
   - causes conflict */
;

/* 8_6 */
if_statement
: IF
  condition
  THEN
  sequence_of_statements
  __ELSIF__THEN__seq_of_stmts__
  _ELSE__seq_of_stmts_
  END_ERR
  IF
  Semicolon_ERR
;


__ELSIF__THEN__seq_of_stmts__
: /*empty*/
| ELSIF
  condition
  THEN
  sequence_of_statements
  __ELSIF__THEN__seq_of_stmts__
{
  yyerrok;
}
;

_ELSE__seq_of_stmts_
: /*empty*/
| ELSE
  sequence_of_statements
{
  yyerrok;
}
;

/* 8_7 */
```

```
case_statement
: CASE
  expression
  IS
  case_statement_alternative
  __case_statement_alternative__
  END_ERR
  CASE
  Semicolon_ERR
;


__case_statement_alternative__
: /*empty*/
| __case_statement_alternative__
  case_statement_alternative
{
yyerrok;
}
;

case_statement_alternative
: WHEN
  choices
  Arrow
  sequence_of_statements
;

/* 8_8 */
/*
**  To avoid shift/reduce conflicts, define rules for labeled/unlabeled loop
**  statement
*/
loop_statement
: a_label
unlabeled_loop_statement
| unlabeled_loop_statement
;

unlabeled_loop_statement
: _iteration_scheme_
  LOOP
  sequence_of_statements
  END_ERR
  LOOP
  _label_
  Semicolon_ERR
;


_iteration_scheme_
```

```
: /*empty*/
| iteration_scheme
;

iteration_scheme
: WHILE
  condition
| FOR
  loop_parameter_specification
;

_label_
: /*empty*/
| label
;


loop_parameter_specification
: Identifier
  IN
  discrete_range
;

/* 8_9 */
next_statement
: NEXT
  _label_
  _WHEN__condition_
  Semicolon_ERR
;


_WHEN__condition_
: /*empty*/
| WHEN
  condition
;

/* 8_10 */
exit_statement
: EXIT
  _label_
  _WHEN__condition_
  Semicolon_ERR
;

/* 8_11 */
return_statement
: RETURN
  _expression_
  Semicolon_ERR
```

```
;


_expression_
: /*empty*/
| expression
;

/* 8_12 */
null_statement
: NULL_
  Semicolon_ERR
;

/*
* chapter 9 - concurrent statements
*/

/* 9_0 */
set_of_statements
: __concurrent_statement__
| error END_ERR
;


__concurrent_statement__
: /*empty*/
| concurrent_statement
  __concurrent_statement__
;

concurrent_statement
: block_statement
| process_statement
    {
       #ifdef _DB1_
       puts("---! arch_add_process()");
       #endif
       arch_add_process();    /* NOTE: Only one process is allowed!!! */
    }
| concurrent_assertion_statement
| concurrent_signal_assignment_statement
| component_instantiation_statement
    {
       #ifdef _DB1_
       puts("---! arch_add_comp_inst_list()");
       #endif
       arch_add_comp_inst_list();
    }
| generate_statement
| concurrent_procedure_call
```

```
;

/* 9_1 */
block_statement    /****** needs changing *******/
: a_label
  BLOCK
  _guard_expression_
  _generic_clause_map_aspect_
  _port_clause_map_aspect_
  block_declarative_part
  BEGIN_
  set_of_statements
  END_ERR
  BLOCK
  _label_
  Semicolon_ERR
;


_guard_expression_
: /*empty*/
| guard_expression
;

guard_expression
: '('
  expression
  RightParen_ERR
;

_generic_clause_map_aspect_
: /*empty*/
| generic_clause
  _generic_map_aspect_Semicolon_
;
_generic_map_aspect_Semicolon_
:
| generic_map_aspect
Semicolon_ERR
;

_port_clause_map_aspect_
: /*empty*/
| port_clause
  _port_map_aspect_Semicolon_
;
_port_map_aspect_Semicolon_
:
| port_map_aspect
Semicolon_ERR
;
```

```
block_declarative_part
: __block_declarative_item__
;


/* 9_2 */
/*
**  To avoid shift/reduce conflicts, define rules for labeled/unlabeled process
**  statements_
*/
process_statement
: a_label
unlabeled_process_statement
| unlabeled_process_statement
;

unlabeled_process_statement
: PROCESS
      {
         #ifdef _DB1_
         puts("~~~! process_clear()");
         #endif
         process_clear();
      }
  _sensitivity_list_
  process_declarative_part
  BEGIN_
     {
        process_clear();
        mcode_clear_list();
     }
  sequence_of_statements
     {
         // Place END opcode in block
         mcode_add(M_END);
         process_add_mcode();
         new_sim_block(arch_c_get_id());
         add_code_to_sim_block(create_c_sim_process(),"CORRECT_CODE_TITLE");
         create_all_hypo();
         finish_sim_block();
     }
  END_ERR
  PROCESS
  _label_
  Semicolon_ERR
;


_sensitivity_list_
: /*empty*/
```

```
| sensitivity_list
;

sensitivity_list
: '('
  signal_list
  RightParen_ERR
;

process_declarative_part
: __process_declarative_item__
;


__process_declarative_item__
: /*empty*/
| __process_declarative_item__
  process_declarative_item
{
yyerrok;
}
;

process_declarative_item
: constant_declaration
| variable_declaration
| type_declaration
| subtype_declaration
| attribute_declaration
| attribute_specification
| subprogram_declaration
| subprogram_body
| file_declaration
| alias_declaration
| use_clause
;

/* 9_3 */
/*
** to avoid shift/reduce conflicts for concurrent_procedure_call
** define rules for labeled and unlabeled statements separately
*/

concurrent_procedure_call
: a_label
  unlabeled_concurrent_procedure_call
| unlabeled_concurrent_procedure_call
;

unlabeled_concurrent_procedure_call
: procedure_call_statement
```

```
;


/* 9_4 */
/*
**  To avoid shift/reduce conflicts, define rules for labeled/unlabeled
**  concurrent_assertion_statements_
**
**  This creates an equivalent process statement which has a sensitivity
**  list of the longest static prefix of each signal name appearing in
**  the boolean expression of the assertion statement_
*/
concurrent_assertion_statement
: a_label
unlabeled_concurrent_assertion_statement
| unlabeled_concurrent_assertion_statement
;

unlabeled_concurrent_assertion_statement
: assertion_statement
;

/* 9_5 */
/*
**  To avoid shift/reduce conflicts, define rules for labeled/unlabeled
**  concurrent_signal_assignment_statements_
**
**  This creates an equivalent process statement_  See 8_2_4 of the LRM_
*/
concurrent_signal_assignment_statement
: a_label
  unlabeled_conditional_signal_assignment
| unlabeled_conditional_signal_assignment
| a_label
  unlabeled_selected_signal_assignment
| unlabeled_selected_signal_assignment
;


/* 9_5_1 */
unlabeled_conditional_signal_assignment
: target
  LESym
  /* options */
  _GUARDED_
  _TRANSPORT_
  /* conditional_waveforms */
  __waveform__WHEN__condition__ELSE__
  waveform
    Semicolon_ERR
```

```
;


__waveform__WHEN__condition__ELSE__
: /*empty*/
| __waveform__WHEN__condition__ELSE__
  waveform
  WHEN
  expression
  ELSE
;

/* 9_5_2 */
unlabeled_selected_signal_assignment
: WITH
  expression
  SELECT
  target
  LESym
  /* options */
  _GUARDED_
  _TRANSPORT_
  /* selected_waveforms */
  waveform
  WHEN
  choices
  ___waveform__WHEN__choices__ /* changed from LRM for consistancy */
  Semicolon_ERR
;


___waveform__WHEN__choices__
: /*empty*/
| ___waveform__WHEN__choices__
  ','
  waveform
  WHEN
  choices
;


_GUARDED_
: /*empty*/
| GUARDED
;

_TRANSPORT_
: /*empty*/
| TRANSPORT
;
```

```
/* 9_6 */
component_instantiation_statement
: a_label
  /* Identifier */
  name
      {

         comp_inst_list_add_node(ident_save_get());
         comp_inst_entity(ident_get());
      }
  _generic_map_aspect_
      {
         portmap_list_clear();
      }
  _port_map_aspect_
      {
         comp_inst_portmap(portmap_get());
      }
  Semicolon_ERR
;


_port_map_aspect_
: /*empty*/
| port_map_aspect
;
port_map_aspect
:
  PORT
  MAP
     {
        assoc_list_clear();
     }
  '('        /* was: PORT aggregate */
  association_list
     {
        portmap_list_add_node( assoc_list_get() );
     }
  RightParen_ERR
;


_generic_map_aspect_
: /*empty*/
| generic_map_aspect
;
generic_map_aspect
:
  GENERIC
    MAP
  '('
```

```
  association_list
  RightParen_ERR
;

/* 9_7 */
/*
**  To avoid shift/reduce conflicts, define labeled/unlabeled generate
**   statements_
*/
generate_statement
: a_label
unlabeled_generate_statement
/* |  unlabeled_generate_statement
*/ ;

unlabeled_generate_statement
: generation_scheme
  GENERATE
  set_of_statements
  END_ERR
  GENERATE
  _label_
  Semicolon_ERR
;

generation_scheme
: FOR
  generate_parameter_specification
| IF
  condition
;

generate_parameter_specification
: Identifier
  IN
  discrete_range
;

condition
: expression
;

/*
**  label declaration
*/
a_label
: label
':'
      {
         ident_save();  /* Save ident; case of 2 idents before parsed */
      }
```

```
;


label
: Identifier
;

/*
**  Chapter 10:  Scope and Visibility
*/

/* 10_4 */
use_clause
: USE
  selected_name    /* package simple name */
  ___selected_name__
  Semicolon_ERR
;


___selected_name__
: /*empty*/
| ___selected_name__
  ','
  selected_name
;

/*
**  Design Units and Their Analysis
*/

/* 11_1 */
design_file
: design_unit
  __design_unit__
;

__design_unit__
: /*empty*/
| design_unit
  __design_unit__
;

design_unit
: context_clause
  library_unit
;

library_unit
: primary_unit
| secondary_unit
```

```
| error
;

primary_unit
: entity_declaration
| configuration_declaration
| package_declaration
;

secondary_unit
: architecture_body
| package_body
;

/* Design Libraries */
/* 11_2 */

library_clause
: LIBRARY
  logical_name_list
  Semicolon_ERR
;

logical_name_list
: Identifier
  ___logical_name__
;

___logical_name__
: /*empty*/
| ___logical_name__
  ','
  Identifier
;

/* 11_3 */
context_clause
: __context_item__
;

__context_item__
: /*empty*/
| __context_item__
  context_item
;

context_item
: library_clause
| use_clause
;
```

```
/* A_4 */
/* abstract_literal */
/*
**  Normally, the grammar for abstract literal would be found here_  It
**  has been moved to the end of this file_  There you will find an
**  explanation_
*/


/*********************************
**
**  Error recovery non-terminals
**
*********************************/

/*
 *  Make ';', ')', and 'end' significant for error recovery_
 */

RightParen_ERR
:  ')'
{
   yyerrok;
}
;

Semicolon_ERR
:  ';'
{
   yyerrok;
}
;

END_ERR
:  END_
{
yyerrok;
}
;

/*
**  In order to implement floating point notation, it was necessary to
**  declare the types of the parameters for 'abstract_literal_real' and
**  'abstract_literal_int', as well as the return type for 'abstract_literal'_
**  But if you do this, then yacc demands that all following grammar rules
**  be similarily typed_  Therefore, this special case was made the last
**  rule in the grammar_  This will cause the compiler to complain about
**  "struct/union or struct/union pointer required", but the source code
**  produced by yacc is correct_
*/
abstract_literal
```

```
: DecimalInt  {/*puts("---! found decimal int");*/}
| DecimalReal {puts("---! found decimal real");}
| BasedInt    {puts("---! found based   int");}
| BasedReal   {puts("---! found based   real");}
;
%%
int yyerror(char *s)
{
printf("YYError:  %s\n", s);
}
```

## *D.3 UV.LEX*

```
%{
/*

    UV.LEX

    This file contains the FLEX code for recognizing the VHDL tokens
*/

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <io.h>
#include "ident.h"
#include "thesis.h"

#include "uv_tab.h"
#include <alloc.h>

char  *strip_underscore();
extern char YACC_STR_LIT[];
extern FILE   *infile;

#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
   { \
   int c = getc(infile); \
   result = (c==EOF) ? YY_NULL : (buf[0]=c,1); \
   }
%}

A       [aA]
B       [bB]
C       [cC]
D       [dD]
E       [eE]
F       [fF]
G       [gG]
H       [hH]
I       [iI]
J       [jJ]
K       [kK]
L       [lL]
M       [mM]
N       [nN]
O       [oO]
P       [pP]
Q       [qQ]
R       [rR]
S       [sS]
T       [tT]
```

```
U         [uU]
V         [vV]
W         [wW]
X         [xX]
Y         [yY]
Z         [zZ]

digit     [0-9][0-9_]*
intlit    {digit}
integer   {intlit}
string    ["]([^\n\t"]|(\"\"))*["]
comment   --[^\n]*
ws        [ \t]+
nl        \n

varasgn      :=
doublestar   [*][*]
lesym        [<][=]
gesym        >=
nesym        \\=
arrow        =>

%%
{ws}      ;
{comment}  {if(is_flag_set(PRINT_COMM)) puts(yytext);} ;
'\'' {
puts("SOMETHING IS RONG");
   return Apostrophe;
}
{doublestar} {
   return DoubleStar;
}

{varasgn} {
  return VarAsgn;
}
{lesym} {
   return LESym;
}
{B}{I}{T} {
   return BIT;
}
{B}{O}{X} {
   return Box;
}
{arrow} {
   return Arrow;
}
{gesym} {
   return GESym;
}
```

```
{B}{A}{R} {
   return Bar;
}
{nesym} {
   return NESym;
}
{S}{L}{A}{S}{H} {
   return Slash;
}
{I}{digit}{digit}{digit} {
/*printf("Lexxed |%s|\n", yytext);*/
ident_set(yytext);
return Identifier;
}
{intlit} {
   lit_set(yytext);
   return DecimalInt;
}
{R}{E}{A}{L} {
   return DecimalReal;
}
{B}{A}{S}{E}{D}{I}{N}{T} {
   return BasedInt;
}
{B}{A}{S}{E}{D}{R}{E}{A}{L} {
   return BasedReal;
}
{C}{H}{A}{R}{A}{C}{T}{E}{R}{L}{I}{T} {
   return CharacterLit;
}
{S}{T}{R}{I}{N}{G}{L}{I}{T} {
   return StringLit;
}
{B}{I}{T}{S}{T}{R}{I}{N}{G}{L}{I}{T} {
   return BitStringLit;
}
{A}{B}{S} {
   return ABS;
}
{A}{C}{C}{E}{S}{S} {
   return ACCESS;
}
{A}{F}{T}{E}{R} {
   return AFTER;
}
{A}{L}{I}{A}{S} {
   return ALIAS;
}
{A}{L}{L} {
   return ALL;
}
```

```
{A}{N}{D} {
   return AND;
}
{A}{R}{C}{H}{I}{T}{E}{C}{T}{U}{R}{E} {
   return ARCHITECTURE;
}
{A}{R}{R}{A}{Y} {
   return ARRAY;
}
{A}{S}{S}{E}{R}{T} {
   return ASSERT;
}
{A}{T}{T}{R}{I}{B}{U}{T}{E} {
   return ATTRIBUTE;
}
{B}{E}{G}{I}{N} {
   return BEGIN_;
}
{B}{L}{O}{C}{K} {
   return BLOCK;
}
{B}{O}{D}{Y} {
   return BODY;
}
{B}{U}{F}{F}{E}{R} {
   return BUFFER;
}
{B}{U}{S} {
   return BUS;
}
{C}{A}{S}{E} {
   return CASE;
}
{C}{O}{M}{P}{O}{N}{E}{N}{T} {
   return COMPONENT;
}
{C}{O}{N}{F}{I}{G}{U}{R}{A}{T}{I}{O}{N} {
   return CONFIGURATION;
}
{C}{O}{N}{S}{T}{A}{N}{T} {
   return CONSTANT;
}
{D}{I}{S}{C}{O}{N}{N}{E}{C}{T} {
   return DISCONNECT;
}
{D}{O}{W}{N}{T}{O} {
   return DOWNTO;
}
{E}{L}{S}{E} {
   return ELSE;
}
```

```
{E}{L}{S}{I}{F} {
   return ELSIF;
}
{E}{N}{D} {
   return END_;
}
{E}{N}{T}{I}{T}{Y} {
   return ENTITY;
}
{E}{X}{I}{T} {
   return EXIT;
}
{F}{I}{L}{E} {
   return FILE_;
}
{F}{O}{R} {
   return FOR;
}
{F}{U}{N}{C}{T}{I}{O}{N} {
   return FUNCTION;
}
{G}{E}{N}{E}{R}{A}{T}{E} {
   return GENERATE;
}
{G}{E}{N}{E}{R}{I}{C} {
   return GENERIC;
}
{G}{U}{A}{R}{D}{E}{D} {
   return GUARDED;
}
{I}{F} {
   return IF;
}
{I}{N}{O}{U}{T} {
   return INOUT;
}
{I}{N} {
   return IN;
}
{I}{S} {
   return IS;
}
{L}{A}{B}{E}{L} {
   return LABEL;
}
{L}{I}{B}{R}{A}{R}{Y} {
   return LIBRARY;
}
{L}{I}{N}{K}{A}{G}{E} {
   return LINKAGE;
}
```

```
{L}{O}{O}{P} {
   return LOOP;
}
{M}{A}{P} {
   return MAP;
}
{M}{O}{D} {
   return MOD;
}
{N}{A}{N}{D} {
   return NAND;
}
{N}{E}{W} {
   return NEW;
}
{N}{E}{X}{T} {
   return NEXT;
}
{N}{O}{R} {
   return NOR;
}
{N}{O}{T} {
   return NOT;
}
{N}{U}{L}{L} {
   return NULL_;
}
{O}{F} {
   return OF;
}
{O}{N} {
   return ON;
}
{O}{P}{E}{N} {
   return OPEN;
}
{O}{R} {
   return OR;
}
{O}{T}{H}{E}{R}{S} {
   return OTHERS;
}
{O}{U}{T} {
   return OUT;
}
{P}{A}{C}{K}{A}{G}{E} {
   return PACKAGE;
}
{P}{O}{R}{T} {
   return PORT;
}
```

```
{P}{R}{O}{C}{E}{D}{U}{R}{E} {
   return PROCEDURE;
}
{P}{R}{O}{C}{E}{S}{S} {
   return PROCESS;
}
{R}{A}{N}{G}{E} {
   return RANGE;
}
{R}{E}{C}{O}{R}{D} {
   return RECORD;
}
{R}{E}{G}{I}{S}{T}{E}{R} {
   return REGISTER;
}
{R}{E}{M} {
   return REM;
}
{R}{E}{P}{O}{R}{T} {
   return REPORT;
}
{R}{E}{T}{U}{R}{N} {
   return RETURN;
}
{S}{E}{L}{E}{C}{T} {
   return SELECT;
}
{S}{E}{V}{E}{R}{I}{T}{Y} {
   return SEVERITY;
}
{S}{I}{G}{N}{A}{L} {
   return SIGNAL;
}
{S}{U}{B}{T}{Y}{P}{E} {
   return SUBTYPE;
}
{T}{H}{E}{N} {
   return THEN;
}
{T}{O} {
   return TO;
}
{T}{R}{A}{N}{S}{P}{O}{R}{T} {
   return TRANSPORT;
}
{T}{Y}{P}{E} {
   return TYPE;
}
{U}{N}{I}{T}{S} {
   return UNITS;
}
```

```
{U}{N}{T}{I}{L} {
   return UNTIL;
}
{U}{S}{E} {
   return USE;
}
{V}{A}{R}{I}{A}{B}{L}{E} {
   return VARIABLE;
}
{W}{A}{I}{T} {
   return WAIT;
}
{W}{H}{E}{N} {
   return WHEN;
}
{W}{H}{I}{L}{E} {
   return WHILE;
}
{W}{I}{T}{H} {
   return WITH;
}
{X}{O}{R} {
   return XOR;
}
{U}{N}{A}{R}{Y}{S}{I}{G}{N} {
   return UNARY_SIGN;
}
{nl}  {
/*   extern int lineno; */
/*   lineno++;*/
/*puts("End of Line.");*/
}

.  return yytext[0];

%%
```

# Appendix E. *Parser Source Code*

## *E.1 Overview*

These modules are tightly linked to the UV module described in Appendix A.1. As soon as a VHDL construct has been parsed, the associated code in the UV module calls the routines in one of the following modules. VHDL constructs that consist of other VHDL objects reference the routines of the sub-object's module.

All routines for the parsed object are in a separate module. In general, each module has an init function, functions to set the values of the object, and a function to return the object. In addition, there are also functions to print the object's value to the screen.

## *E.2 ARCH.H*

```
/*
   VHDL PARSER

   File: ARCH.H

   Date: 7 July 1992

   This module handles the creation of an ARCHITECTURE by the
   VHDL parser. These routines are called by the BISON program.

   Routines:
   ---------
arch_clear() -- Clear current arch settings
arch_id()    -- Add current identitier to arch
   arch_name()  -- Add current ident as arch name
   arch_add_signal_list() -- Add current signal list to current arch
   arch_print( ent) -- Print specified architecture
*/
/*-----------------------------------------------------------------*/
#ifndef __arch_h__
#define __arch_h__ 1

#ifndef __ident_h__
#include "ident.h"
#endif

#ifndef __signal_h__
```

```
#include "signal.h"
#endif

#ifndef __comp_h__
#include "comp.h"
#endif

#ifndef __comp_in_h__
#include "comp_in.h"
#endif

#ifndef __process_h__
#include "process.h"
#endif
/*-----------------------------------------------------------------*/
typedef struct S_arch {
   T_ident        id;    /* Instance of Arch */
   T_ident        name;  /* entity for arch */
   TP_signal      signal_list;  /* List of local signals */
   TP_comp        comp_list;    /* List of local comp declarations */
   TP_comp_inst   comp_inst_list;  /* List of comp instantiations */
   T_process      process;      /* the ONE process in the architecture */
   struct S_arch *next;
} T_arch;

/* arch node pointer */
typedef T_arch *  TP_arch;

/*-----------------------------------------------------------------*/

/* prototypes */
void arch_clear(void);
void arch_add(void);
void arch_name(void);
void arch_add_signal_list(void);
void arch_add_comp_list(void);
void arch_add_comp_inst_list(void);
void arch_add_process(void);
void arch_c_list_print(void);
void arch_list_print(TP_arch list);
void arch_print(TP_arch list);
int  arch_c_get_id(void);
TP_arch arch_get(int id);

/*-----------------------------------------------------------------*/
#endif
```

*E.3 ARCH.CPP*

```
/*
   VHDL PARSER

   File: ARCH.C

   Date: 7 July 1992

   This module handles the creation of an ARCHITECTURE by the
   VHDL parser. These routines are called by the BISON program.

*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <malloc.h>
#include <string.h>
#include "thesis.h"
#include "arch.h"
/*-------------------------------------------------------------------*/
/* Current arch being created */

static TP_arch current_arch_list;

/*-------------------------------------------------------------------*/
void arch_clear(void)
{
   current_arch_list = NULL;
}
void arch_add(void)
{
   TP_arch new_node;

   if( (new_node = (TP_arch)(malloc(sizeof(T_arch)))) == NULL ) {
      yyerror("Out of memory in ARCH_ADD()");
      exit(129);
   }
   new_node->id             = ident_get();
   new_node->signal_list    = NULL;
   new_node->comp_list      = NULL;
   new_node->comp_inst_list = NULL;
   new node->process        = NULL_PROCESS;
   new_node->next           = current_arch_list;
   current_arch_list        = new_node;
}

// Add name to architecture
void arch_name(void)
{
   current_arch_list->name = ident_get();
```

```
}

void arch_add_signal_list(void)
{
   current_arch_list->signal_list = signal_get();
}

void arch_add_comp_list(void)
{
   current_arch_list->comp_list = comp_get();
}

void arch_add_comp_inst_list(void)
{
   current_arch_list->comp_inst_list = comp_inst_list_get();
}

void arch_add_process(void)
{
   current_arch_list->process = process_get();
}

int  arch_c_get_id(void)
{
   return current_arch_list->id;
}

void arch_c_list_print(void)
{
   arch_print(current_arch_list);
}

void arch_list_print(TP_arch list)
{
   if(list == NULL) {
      puts("Empty arch signal list");
   }
   else {
      while(list != NULL ) {
         arch_print(list);
         list = list->next;
      }
   }
}

void arch_print(TP_arch node)
{
   if(node == NULL) {
      puts("Empty arch signal node");
   }
   else {
```

```
      printf("Arch id: ");
      ident_print( &(node->id) );
      printf(" Name: ");
      ident_print( &(node->name) );
      puts("");
      puts("Arch signal list");
      signal_print( node->signal_list);
      puts("Arch comp list");
      comp_print( node->comp_list);
      puts("Arch component instantiation list");
      comp_inst_list_print( node->comp_inst_list);
      puts("Arch process");
      process_print( &(node->process) );
   }
}

// Return pointer to arch specified by ID
// Return NULL if not found
TP_arch arch_get(int id)
{
   TP_arch ptr = current_arch_list;
   while(ptr != NULL && ptr->id != id) {
      ptr = ptr->next;
   }
   assert(ptr != NULL);
   return ptr;
}

/*-----------------------------------------------------------------------*/
```

*E.4 ASSOC.H*

```
/*
   VHDL PARSER

   File: ASSOC.H

   Date: 9 July 1992

   This module handles the creation of a assoc's list of
   input/output signals. Theses functions are called by the BISON
   program.

   Routines:
   ---------
   assoc_clear()       -- Clear current assoc list
   assoc_add_id()      -- Add current ident to current assoc list !!! NOT USED
   assoc_add_id_list() -- Add current ident_list to current assoc list
   assoc_print(list)   -- Print supplied assoc list. If CURRENT_LIST,
                          print out the current list.
   assoc_get()         -- Get pointer to current list.
*/
/*-------------------------------------------------------------------*/
#ifndef __assoc_h__
#define __assoc_h__ 1

#ifndef __ident_h__
#include "ident.h"
#endif


/* assoc signal node */
typedef struct S_assoc {
   T_ident        left;
   T_ident        right;
   struct S_assoc *next;
} T_assoc;

/* assoc signal node pointer */
typedef T_assoc *  TP_assoc;

/*-------------------------------------------------------------------*/

/* prototypes */
void assoc_list_clear(void);
void assoc_list_free(TP_assoc list);
void assoc_list_add_node(void);
void assoc_left(T_ident left);
void assoc_right(T_ident right);
void assoc_list_print(TP_assoc list);
TP_assoc assoc_list_get(void);
/*-------------------------------------------------------------------*/
```

```
#endif
```

*E.5 ASSOC.CPP*

```
/*
   VHDL PARSER

   File: ASSOC.C

   Date: 9 July 1992

   This module handles the creation of a assoc's list of
   input/output signals. Theses functions are called by the BISON
   program.

*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "thesis.h"
#include "assoc.h"
#include "misc.h"
/*-------------------------------------------------------------------*/
/* Storage for current assoc signal list */

static TP_assoc assoc_signal_list = NULL;

/*-------------------------------------------------------------------*/
void assoc_list_clear(void)
{
   assoc_signal_list = NULL;
}

void assoc_list_free(TP_assoc list)
{
   TP_assoc ptr;

   if(list == CURRENT_LIST) {
      list = assoc_signal_list;
   }
   while(list != NULL) {
      ptr = list->next;
      free(list);
      list = ptr;
   }
}

void assoc_list_add_node(void)
{
   TP_assoc new_node;

   if( (new_node = (TP_assoc)(malloc(sizeof(T_assoc)))) == NULL ) {
      yyerror("Out of memory in assoc_ADD_ID()");
```

```
      exit(101);
   }
   new_node->left     = -1;
   new_node->right    = -1;
   new_node->next     = assoc_signal_list;
   assoc_signal_list  = new_node;
}

void assoc_left(T_ident left)
{
   assoc_signal_list->left = left;
}

void assoc_right(T_ident right)
{
   assoc_signal_list->right = right;
}


void assoc_list_print(TP_assoc list)
{
   if(list == NULL) {
      puts("Empty association list");
   }
   else {
      if(list == CURRENT_LIST ) {
         puts("Found CL; printing current list:");
         list = assoc_signal_list;
      }
      while(list != NULL ) {
         printf("%d => %d\n",
            list->left,list->right);
         list = list->next;
      }
   }
}

TP_assoc assoc_list_get(void)
{
   return assoc_signal_list;
}
/*-----------------------------------------------------------------------*/
```

*E.6 COMP.H*

```
/*
   VHDL PARSER

   File: COMP.H

   Date: 9 July 1992

   This module handles the identifiers encountered by BISON.

   Routines:
   ---------
   comp_clear()         - Clear current comp
   comp_add_id()        - Add ident to comp !!!not used
   comp_add_id_list()   - Add current ident list to current comp
   comp_print()         - Print comp data
   comp_get()           - Get current comp data
*/
/*-----------------------------------------------------------------------*/
#ifndef __comp_h__
#define __comp_h__ 1

#ifndef __ident_h__
#include "ident.h"
#endif

#ifndef __port_h__
#include "port.h"
#endif


/* comp signal node */
typedef struct S_comp {
   T_ident        id;
   TP_port        port;
   struct S_comp *next;
} T_comp;

/* comp signal node pointer */
typedef T_comp *  TP_comp;

/*-----------------------------------------------------------------------*/

/* prototypes */
void comp_clear_list(void);
void comp_add_comp(void);
void comp_add_port(void);
void comp_print(TP_comp list);
TP_comp comp_get(void);
TP_comp comp_get(int name, TP_comp ptr);
/*-----------------------------------------------------------------------*/
```

```
#endif
```

*E.7 COMP.CPP*

```
/*
   VHDL PARSER

   File: COMP.C

   Date: 2 July 1992

   This module handles the creation of a comp's list of
   input/output signals. Theses functions are called by the BISON
   program.

*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "thesis.h"
#include "comp.h"
#include "misc.h"
/*-------------------------------------------------------------------------*/
/* Storage for current comp signal list */

static TP_comp comp_list = NULL;

/* Storage for current comp */
static T_comp current_comp;

/*-------------------------------------------------------------------------*/
void comp_clear_list(void)
{
   comp_list = NULL;
}

void comp_add_comp(void)
{
   TP_comp new_node;

   if( (new_node = (TP_comp)(malloc(sizeof(T_comp)))) == NULL ) {
      yyerror("Out of memory in comp_ADD_ID()");
      exit(102);
   }
   new_node->id      = ident_get();
   new_node->next    = comp_list;
   comp_list  = new_node;
}

void comp_add_port(void)
{
   comp_list->port = port_get();
}
```

```
void comp_print(TP_comp list)
{
   if(list == NULL) {
      puts("Empty comp signal list");
   }
   else {
      if(list == CURRENT_LIST ) {
         puts("Found CL; printing current list:");
         list = comp_list;
      }
      while(list != NULL ) {
         printf("Name: ");
         ident_print( &(list->id) );
         puts("");
         printf("Port list:\n");
         port_print( (list->port) );
         list = list->next;
      }
   }
}

TP_comp comp_get(void)
{
   return comp_list;
}

// Return pointer to entity specified by ID
// Return NULL if not found
TP_comp comp_get(int id, TP_comp ptr)
{
   while(ptr != NULL && ptr->id != id) {
      ptr = ptr->next;
   }
   return ptr;
}
/*-----------------------------------------------------------------*/
```

*E.8 COMP_IN.H*

```
/*
   VHDL PARSER

   File: COMP_IN.H

   Date: 9 July 1992

   This module handles the creation of a comp_inst's list of
   input/output signals. Theses functions are called by the BISON
   program.

   Routines:
   ---------
   comp_inst_clear()       -- Clear current comp_inst list
   comp_inst_add_id()      -- Add current ident to current comp_inst list
   comp_inst_add_id_list() -- Add current ident_list to current comp_inst list
   comp_inst_print(list)   -- Print supplied comp_inst list. If CURRENT_LIST,
                           print out the current list.
   comp_inst_get()         -- Get pointer to current list.
*/
/*-----------------------------------------------------------------*/
#ifndef __comp_in_h__
#define __comp_in_h__ 1

#ifndef __ident_h__
#include "ident.h"
#endif

#ifndef __portmap_h__
#include "portmap.h"
#endif


/* comp_inst signal node */
typedef struct S_comp_inst {
   T_ident        name;
   T_ident        entity;
   TP_portmap     portmap;
   struct S_comp_inst *next;
} T_comp_inst;

/* comp_inst signal node pointer */
typedef T_comp_inst *  TP_comp_inst;

/*-----------------------------------------------------------------*/

/* prototypes */
void comp_inst_list_clear(void);
void comp_inst_list_free(TP_comp_inst list);
void comp_inst_list_add_node(T_ident name);
```

```
void comp_inst_entity(T_ident entity);
void comp_inst_portmap(TP_portmap portmap);
void comp_inst_list_print(TP_comp_inst list);
TP_comp_inst comp_inst_list_get(void);
TP_comp_inst comp_inst_get(int id, TP_comp_inst ptr);

/*-----------------------------------------------------------------------*/
#endif
```

## *E.9 COMP_IN.CPP*

```
/*
   VHDL PARSER

   File: COMP_IN.C

   Date: 9 July 1992

   This module handles the creation of a comp_inst's list of
   input/output signals. Theses functions are called by the BISON
   program.

*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "thesis.h"
#include "comp_in.h"
#include "misc.h"
/*-------------------------------------------------------------------*/
/* Storage for current comp_inst signal list */

static TP_comp_inst comp_inst_list = NULL;

/*-------------------------------------------------------------------*/
void comp_inst_list_clear(void)
{
   comp_inst_list = NULL;
}

void comp_inst_list_free(TP_comp_inst list)
{
   TP_comp_inst ptr;

   if(list == CURRENT_LIST) {
      list = comp_inst_list;
   }
   while(list != NULL) {
      ptr = list->next;
      free(list);
      list = ptr;
   }
}

void comp_inst_list_add_node(T_ident name)
{
   TP_comp_inst new_node;

   if( (new_node = (TP_comp_inst)(malloc(sizeof(T_comp_inst)))) == NULL ) {
      yyerror("Out of memory in comp_inst_ADD_ID()");
```

```
      exit(103);
   }
   new_node->name  = name;
   new_node->next  = comp_inst_list;
   comp_inst_list  = new_node;
}

void comp_inst_entity(T_ident entity)
{
   comp_inst_list->entity = entity;
}


void comp_inst_portmap(TP_portmap portmap)
{
   comp_inst_list->portmap = portmap;
}

void comp_inst_list_print(TP_comp_inst list)
{
   if(list == NULL) {
      puts("Empty comp_instiation list");
   }
   else {
      if(list == CURRENT_LIST ) {
         puts("Found CL; printing current component list:");
         list = comp_inst_list;
      }
      while(list != NULL ) {
         printf("Name: %2d   Entity: %2d\n",
            list->name,list->entity);
         puts("Port map list:");
         portmap_list_print(list->portmap);
         list = list->next;
      }
   }
}

TP_comp_inst comp_inst_list_get(void)
{
   return comp_inst_list;
}
/*-----------------------------------------------------------------*/
// Return pointer to comp_inst specified by name in supplied comp_inst list
// Return NULL if not found
TP_comp_inst comp_inst_get(int name, TP_comp_inst ptr)
{
   while(ptr != NULL && ptr->name != name) {
      ptr = ptr->next;
   }
   return ptr;
```

```
}
/*-----------------------------------------------------------------*/
```

## *E.10 MISC.CPP*

```
/*
   VHDL PARSER

   File: MISC.C

   Date: 2 July 1992

   Miscellaneous routines for functions called from BISON

*/

#include "thesis.h"
#include "misc.h"
/*-------------------------------------------------------------------*/
/* Storage for current values */

static int current_port_direction;
static int current_type_mark;

/*-------------------------------------------------------------------*/
/* Routines for handling current port signal direction */

void direct_set(int new_dir)
{
   current_port_direction = new_dir;
}

int direct_get()
{
   return current_port_direction;
}
/*-------------------------------------------------------------------*/
/* Routines for handling current type mark */

void type_set(int new_type)
{
   current_type_mark = new_type;
}

int type_get()
{
   return current_type_mark;
}
/*-------------------------------------------------------------------*/
```

## *E.11 ENTITY.H*

```
/*
   VHDL PARSER

   File: ENTITY.H

   Date: 2 July 1992

   This module handles the creation of an ENTITY by the
   VHDL parser. These routines are called by the BISON program.

   Routines:
   ---------
entity_clear() -- Clear current entity settings
entity_id()    -- Add current identitier to entity
entity_add_port() - Add current port list to entity
   entity_print()    - Print out specified entity - NULL prints
                       out current entity
*/

/*-------------------------------------------------------------------*/
#ifndef __entity_h__
#define __entity_h__ 1

#ifndef __ident_h__
#include "ident.h"
#endif

#ifndef __port_h__
#include "port.h"
#endif
/*-------------------------------------------------------------------*/

typedef struct S_entity {
   T_ident          id;
   TP_port          port_list;
   struct S_entity *next;
} T_entity;

/* Entity node pointer */
typedef T_entity *  TP_entity;

/*-------------------------------------------------------------------*/
/* prototypes */

void entity_list_clear(void);
void entity_add(void);
void entity_add_port(void);
void entity_c_list_print(void);
void entity_list_print(TP_entity list);
void entity_print(TP_entity list);
```

```
TP_entity entity_get(int id);

/*-------------------------------------------------------------------------*/

#endif
```

*E.12 ENTITY.CPP*

```
/*
   VHDL PARSER

   File: ENTITY.C

   Date: 2 July 1992

   This module handles the creation of an ENTITY by the
   VHDL parser. These routines are called by the BISON program.

*/
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <alloc.h>
#include <string.h>
#include "thesis.h"
#include "entity.h"
/*-------------------------------------------------------------------*/
/* Current entity being created */

// static T_entity current_entity;
static TP_entity current_entity_list;
/*-------------------------------------------------------------------*/
void entity_list_clear(void)
{
   current_entity_list = NULL;
}

void entity_add(void)
{
   TP_entity new_node;

   if( (new_node = (TP_entity)(malloc(sizeof(T_entity)))) == NULL ) {
      yyerror("Out of memory in ENTITY_ADD()");
      exit(104);
   }
   new_node->id        = ident_get();
   new_node->port_list = NULL;
   new_node->next      = current_entity_list;
   current_entity_list = new_node;
}

void entity_add_port(void)
{
   current_entity_list->port_list = port_get();
}

void entity_c_list_print(void)
{
```

```
   entity_print(current_entity_list);
}
void entity_list_print(TP_entity list)
{
   if(list == NULL) {
      puts("Empty entity signal list");
   }
   else {
      while(list != NULL ) {
         entity_print(list);
         list = list->next;
      }
   }
}

void entity_print(TP_entity list)
{
   if(list == NULL) {
      puts("Empty entity signal list");
   }
   else {
      printf("Entity id: ");
      ident_print( &(list->id) );
      puts("");
      puts("Entity signal list");
      port_print( list->port_list);
      puts("----------------------------");
   }
}

// Return pointer to entity specified by ID
// Return NULL if not found
TP_entity entity_get(int id)
{
   TP_entity ptr = current_entity_list;
   while(ptr != NULL && ptr->id != id) {
      ptr = ptr->next;
   }
   assert(ptr != NULL);
   return ptr;
}

/*-------------------------------------------------------------------*/
```

## *E.13 GENERATE.H*

```
//
//
// File: GENERATE.H
//
//
// 23 July 1992
//
//-------------------
#ifndef __generate_h__
#define __generate_h__

/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
/* Prototypes */
void generate_entity_name(int entity_name);
void generate_arch_name(int arch_id);
void generate_ident_list(void);

generate_got_top_id(int top_id);
generate_got_top_entity_id(int id);
generate_got_top_arch_id(int id);

#endif
```

## E.14 GENERATE.CPP

```
//
//
// File: GENERATE.C
//
//
// 22 July 1992
//
// This module handles the mcode generation for Calvin.
//-----------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include "thesis.h"
#include "entity.h"
#include "arch.h"
#include "comp_in.h"
#include "misc.h"
#include "behave.hpp"
#include "signal.hpp"
#include "comsen.hpp"
/*-----------------------------------------------------------------*/
// Static variables

static int top_id;                  // Name of top configuration
static int top_entity_id;           // Entity  for top-level circuit
static TP_entity top_entity;        // Pointer to  top-level entity
static int top_arch_id;             // Arch    for top-level circuit
static TP_arch top_arch;            // Pointer to  top-level arch
static int current_config_id;       // Current configuration id

static int save_ent_id;       // Temp storage for current entity
static int save_arch_id;      // Temp storage for current arch

TP_port  G_commands;
TP_port  G_sensors;
/*-----------------------------------------------------------------*/
void so_far(void);
/*-----------------------------------------------------------------*/
// Save name of configuration
generate_got_top_id(int id)
{
   // printf("--- CONFIGURATION %2d",id);
   top_id = id;
}
/*-----------------------------------------------------------------*/
// Save name of entity for top configuration
generate_got_top_entity_id(int id)
{
   SignalRecord sr;
```

```
   char  title[MAX_NAME_SIZE+10];

   top_entity_id = id;
   top_entity    = entity_get(id);

   TP_port port_list = top_entity->port_list;
   while(port_list != NULL) {
      switch( port_list->direction ) {
      case V_IN:
         sprintf(title,"IN__#%d",port_list->id);
         sr = SignalRecord(port_list->id,title,COMMAND_SR);
         add_command(port_list->id);
         add_signal_rec(sr);
         break;
      case V_OUT:
         sprintf(title,"OUT_#%d",port_list->id);
         sr = SignalRecord(port_list->id,title,SENSOR_SR);
         add_sensor(port_list->id);
         add_signal_rec(sr);
         break;
      default:
         yyerror("Illegal direction in generate_com_sig");
      }
      port_list = port_list->next;
   }
}
/*-------------------------------------------------------------------*/
// Save name for top arch
generate_got_top_arch_id(int id)
{
   SignalRecord sr;

   top_arch_id = id;
   top_arch    = arch_get(id);

   TP_signal signal_list = top_arch->signal_list;
   while(signal_list != NULL) {
      sr = SignalRecord(signal_list->id,"x",-987);
      add_signal_rec(sr);
      signal_list = signal_list->next;
   }
}
/*-------------------------------------------------------------------*/
// Found entity name to instantiate with
void generate_entity_name(int entity_name)
{
   save_ent_id = entity_name;
}
/*-------------------------------------------------------------------*/
// found arch name to instantiate with
void generate_arch_name(int arch_id)
```

```
{
   save_arch_id = arch_id;
}
/*-------------------------------------------------------------------*/
void create_behave( int behave_id, TP_arch arch_ptr )
{
   Behave       bb;
   SignalRecord sr_ptr;

   bb = Behave(behave_id);
   bb.set_block_id(arch_ptr->id);

   TP_comp_inst this_comp = comp_inst_get(behave_id, top_arch->comp_inst_list);
   TP_portmap   portmap   = this_comp->portmap;
   TP_assoc     assoc     = portmap->assoc_list;
   while( assoc != NULL ) {
      // Find comp that matches this comp
      TP_comp comp = comp_get( this_comp->entity, top_arch->comp_list);
      TP_port port = port_get( assoc->left, comp->port);
      sr_ptr = get_signal_rec(assoc->right);
      switch( port->direction) {
      case V_IN:
         bb.add_input(assoc->right,port->number);
         sr_ptr.add_conns(behave_id);
         break;
      case V_OUT:
         bb.add_output(assoc->right,port->number);
         sr_ptr.set_driver_bi(behave_id);
         break;
      default:
         yyerror("Illegal direction in (generate)create_behave");
      }
      mod_signal_rec(sr_ptr);
      assoc = assoc->next;
   }
   add_behave_inst(bb);
}
/*-------------------------------------------------------------------*/

void generate_ident_list(void)
{
   TP_arch       arch_ptr   = arch_get(   save_arch_id );
   TP_ident_list id_ptr     = ident_list_get();

   while( id_ptr != NULL ) {
      create_behave( id_ptr->id, arch_ptr );
      id_ptr = id_ptr->next;
   }
}
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
```

*E.15 IDENT.H*

```
/*#define __DEBUG__*/
/*
   VHDL PARSER

   File: IDENT.H

   Date: 2 July 1992

   This module handles the identifiers encountered by BISON.

   Routines:
   ---------
   ident_list_clear()-- Clear current identifier list
   ident_list_free() -- Free memory of supplied ident list ptr
   ident_list_add()  -- Add current ident to current identifier list
   ident_list_print()-- Print supplied ident list. If CURRENT_LIST,
                        prints out current ident list.
   ident_list_get()  -- Return pointer to current ident list
   ident_set()       -- Set identifier to supplied string
   ident_get()       -- Return hash value of current identifier
   ident_print( )    -- Print out supplied identifier

*/
/*-------------------------------------------------------------------*/
#ifndef __ident_h__
#define __ident_h__ 1

typedef  int T_ident;


/* Ident node */
typedef struct S_ident {
   T_ident        id;
   struct S_ident *next;
} T_ident_list;

/* Ident node pointer */
typedef T_ident_list *  TP_ident_list;


/* prototypes */
void        ident_list_clear(void);
void        ident_list_free(TP_ident_list list);
void        ident_list_add(void);
void        ident_list_print(TP_ident_list list);
TP_ident_list ident_list_get(void);
void        ident_set(char *s);
T_ident     ident_get(void);
void        ident_print( T_ident *id );
void        ident_save(void);
```

```
T_ident     ident_save_get(void);
void        ident_c_list_print();
void        ident_c_list_free();

void        lit_set(char *str);
int         lit_get(void);

#define MAX_IDENT_LEN 32 /* Maximum length of an identifier */


/*-----------------------------------------------------------------*/
#endif
```

*E.16 IDENT.CPP*

```
/**define __DEBUG__*/
/*
   VHDL PARSER

   File: IDENT.C

   Date: 2 July 1992

   This module handles the identifiers encountered by BISON.

*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <alloc.h>
#include "thesis.h"
#include "ident.h"
/*-------------------------------------------------------------------*/
/* Storage for current identifier */

static char identifier[MAX_IDENT_LEN+1];

static T_ident id_value;
static T_ident save_ident;

/* Storage for current identifier list */

static TP_ident_list ident_signal_list = NULL;

/* Storage for current integer lit */
static int int_lit_value = -1;

/*-------------------------------------------------------------------*/
void ident_list_clear(void)
{
   ident_signal_list = NULL;
}

void ident_c_list_free()
{
   ident_list_free(ident_signal_list);
}

void ident_list_free(TP_ident_list list)
{
   TP_ident_list ptr;

   while(list != NULL) {
      ptr = list->next;
      free(list);
```

```
      list = ptr;
   }
}


void ident_list_add(void)
{
   TP_ident_list new_node;

   if( (new_node = (TP_ident_list)(malloc(sizeof(T_ident_list)))) == NULL ) {
      yyerror("Out of memory in IDENT_ADD_ID()");
      exit(105);
   }

   new_node->id      = ident_get();
   new_node->next    = ident_signal_list;
   ident_signal_list = new_node;
}

void ident_c_list_print()
{
   puts("Printing current list:");
   ident_list_print(ident_signal_list);
}

void ident_list_print(TP_ident_list list)
{
   if(list == NULL) {
      puts("Empty ident signal list");
   }
   else {
      while(list != NULL ) {
         ident_print( &(list->id) );
         printf(" ");
         list = list->next;
      }
   }
}

TP_ident_list ident_list_get(void)
{
   return ident_signal_list;
}


/*-----------------------------------------------------------------*/
void ident_set(char *s)
{
   strncpy(identifier, s, MAX_IDENT_LEN);
   identifier[MAX_IDENT_LEN] = '\0';
```

```
   id_value = atoi(identifier+1);
}

T_ident ident_get(void)
{
   id_value = atoi(identifier+1);
   return id_value;
}

void ident_print( T_ident *id )
{
   printf("%d",*id);
}

void ident_save(void)
{
   save_ident = id_value;
}

T_ident ident_save_get(void)
{
   return save_ident;
}


/*--------------------------------------------------------------------*/
void lit_set(char *str)
{
   int_lit_value = atoi(str);
}

int lit_get(void)
{
   return int_lit_value;
}
/*--------------------------------------------------------------------*/
```

## *E.17 MCODE.H*

```
/*
   VHDL PARSER

   File: MCODE.H

   Date: 7 July 1992

   Routines:
   ---------
   mcode_clear_list()  -- Clear current mcode list pointer
   mcode_add_id()      -- Add current id to current mcode list !!!NOT USED
   mcode_add_id_list() -- Add current id_list to current mcode list
   mcode_print(list)   -- Print specified mcode list. If CURRENT_LIST,
                          print current mcode list.
   mcode_get()         -- Get pointer to current mcode list.
*/
/*-------------------------------------------------------------------*/
#ifndef __mcode_h__
#define __mcode_h__ 1

#ifndef __ident_h__
#include "ident.h"
#endif


/* Port mcode node */
typedef struct S_mcode {
   int             code;
   struct S_mcode *prev;
   struct S_mcode *next;
} T_mcode;

/* Port mcode node pointer */
typedef T_mcode *  TP_mcode;

/*-------------------------------------------------------------------*/
/* Codes for MCODE commands */
// !!!!!! MERGE WITH MCODE.H !!!!!!!

#define M_NULL  -1    // Null opcode
#define M_GET   -2    // Get signal  (signal #)
#define M_POST  -3    // Post signal (signal #, value, delta time)
#define M_PUSH  -4    // Push??
#define M_NOT   -5    // NOT (value)
#define M_AND   -6    // AND (value1, value2)
#define M_OR    -7    // OR  (value1, value2)
#define M_XOR   -8    // XOR (value1, value2)
#define M_END   -9    // End execution
#define M_NAND  -10
#define M_NOR   -11
```

```
#define M_POP   -12   // Pop and discard top value on stack
#define M_STORE -13   // Store (addr) -- Place TOS in temp store
#define M_RETRV -14   // Retrieve (addr) -- Place value from store on TOS

/*-----------------------------------------------------------------*/

/* prototypes */
void     mcode_clear_list(void);
void     mcode_list_free(TP_mcode list);
void     mcode_add(int new_code);
int      mcode_pop(TP_mcode *mod_list);
void     mcode_pop_top(TP_mcode *list);
TP_mcode mcode_begin(TP_mcode list);
void     mcode_print(TP_mcode list);
TP_mcode mcode_get(void);
void     mcode_c_pop_top(void);
void     mcode_c_list_free(void);
int      mcode_c_pop(void);
TP_mcode mcode_c_begin(void);
void     mcode_c_print(void);

/*-----------------------------------------------------------------*/
#endif
```

*E.18 MCODES.CPP*

```
/*
   VHDL PARSER

   File: mcode.C

   Date: 7 July 1992

   This module handles the microcode generation.

*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "thesis.h"
#include "mcode.h"
#include "misc.h"
/*-------------------------------------------------------------------*/
/* Storage for current mcode mcode list */

static TP_mcode mcode_list = NULL;

/*-------------------------------------------------------------------*/
void mcode_clear_list(void)
{
   mcode_list = NULL;
}

void mcode_c_list_free(void)
{
   mcode_list_free(mcode_list);
}

void mcode_list_free(TP_mcode list)
{
   TP_mcode ptr;

   while(list != NULL) {
      ptr = list->prev;
      free(list);
      list = ptr;
   }
}


void mcode_add(int new_code)
{
   TP_mcode new_node;

   if( (new_node = (TP_mcode)(malloc(sizeof(T_mcode)))) == NULL ) {
```

```
      yyerror("Out of memory in PORT_ADD_ID()");
      exit(123);
   }
   new_node->code    = new_code;
   new_node->prev    = mcode_list;
   new_node->next    = NULL;
   if(mcode_list != NULL) {
      mcode_list->next  = new_node;
   }
   mcode_list        = new_node;
}

int mcode_c_pop(void)
{
   return mcode_pop(&mcode_list);
}

int mcode_pop(TP_mcode *mod_list)
{
   TP_mcode list = *mod_list;
   TP_mcode old_node;
   int      node_value;

   if(list == NULL ) {
      return ERROR;
   }

   /* Pop top node off list */
   old_node   = list;
   list       = list->prev;
   if( list != NULL) {
      list->next = NULL;
   }
   /* Return changed list */
   if(mod_list == CURRENT_LIST) {
      mcode_list = list;
   }
   else {
      *mod_list = list;
   }

   /* Get value of node */
   node_value = old_node ->code;

   /* Free the old node */
   free(old_node);

   return node_value;
}

/* pop and discard top of list   */
```

```
void mcode_c_pop_top(void)
{
   mcode_c_pop();
}

void mcode_pop_top(TP_mcode *list)
{
   mcode_pop(list);
}

TP_mcode mcode_c_begin(void)
{
   return mcode_begin(mcode_list);
}

TP_mcode mcode_begin(TP_mcode list)
{
   if(list == NULL) {
      return NULL;
   }
   while(list->prev != NULL ) {
      list = list->prev;
   }
   return list;
}

void mcode_c_print(void)
{
   puts("Printing current mcode list:");
   mcode_print(mcode_list);
}

void mcode_print(TP_mcode list)
{
   if(list == NULL) {
      puts("Empty mcode mcode list");
   }
   else {
      list = mcode_begin(list);
      while(list != NULL ) {
         switch(list->code) {
         case M_GET:
            puts("Get_signal");
            break;
         case M_POST:
            puts("Post");
            break;
         case M_AND:
            puts("And");
            break;
         case M_OR:
```

```
            puts("Or");
            break;
         case M_NAND:
            puts("Nand");
            break;
         case M_NOR:
            puts("Nor");
            break;
         case M_XOR:
            puts("Xor");
            break;
         case M_NOT:
            puts("Not");
            break;
         case M_END:
            puts("End");
            break;
         case M_POP:
            puts("Pop");
            break;
         case M_STORE:
            puts("Store");
            break;
         case M_RETRV:
            puts("Retrieve");
            break;
         default:
            printf(":%d\n",list->code);
         }
         list = list->next;
      }
   }
}

TP_mcode mcode_get(void)
{
   return mcode_c_begin();
}
/*-------------------------------------------------------------------*/
```

*E.19 MISC.H*

```
/*
   VHDL PARSER

   File: MISC.H

   Date: 2 July 1992

   Miscellaneous routines for functions called from BISON

   Routines:
   ---------
   direct_set() -- Set parsed  direction
   direct_get() -- Get current direction
   type_set()   -- Set current type
   type_get()   -- Get current type
*/
/*-------------------------------------------------------------------*/
#ifndef __misc_h__
#define __misc_h__ 1

/*-------------------------------------------------------------------*/
/* Constants */

#define V_IN     1  /* Port direction IN    */
#define V_OUT    2  /* Port direction OUT   */
#define V_BIT    3  /* Type mark BIT        */
/*-------------------------------------------------------------------*/

/* prototypes */
#ifdef __BORLANDC__
void  direct_set(int new_dir);
int   direct_get(void);
void  type_set(int new_type);
int   type_get(void);
#else
void  direct_set();
int   direct_get();
void  type_set();
int   type_get();
#endif

/*-------------------------------------------------------------------*/
#endif
```

## *E.20 MISC.CPP*

```
/*
   VHDL PARSER

   File: MISC.C

   Date: 2 July 1992

   Miscellaneous routines for functions called from BISON

*/

#include "thesis.h"
#include "misc.h"
/*-------------------------------------------------------------------*/
/* Storage for current values */

static int current_port_direction;
static int current_type_mark;

/*-------------------------------------------------------------------*/
/* Routines for handling current port signal direction */

void direct_set(int new_dir)
{
   current_port_direction = new_dir;
}

int direct_get()
{
   return current_port_direction;
}
/*-------------------------------------------------------------------*/
/* Routines for handling current type mark */

void type_set(int new_type)
{
   current_type_mark = new_type;
}

int type_get()
{
   return current_type_mark;
}
/*-------------------------------------------------------------------*/
```

*E.21 PORT.H*

```
/*
   VHDL PARSER

   File: PORT.H

   Date: 2 July 1992

   This module handles the creation of a port's list of
   input/output signals. Theses functions are called by the BISON
   program.

   Routines:
   ---------
   port_clear()       -- Clear current port list
   port_add_id()      -- Add current ident to current port list !!! NOT USED
   port_add_id_list() -- Add current ident_list to current port list
   port_print(list)   -- Print supplied port list. If CURRENT_LIST,
                         print out the current list.
   port_get()         -- Get pointer to current list.
*/
/*-------------------------------------------------------------------*/
#ifndef __port_h__
#define __port_h__ 1

#ifndef __ident_h__
#include "ident.h"
#endif


/* Port signal node */
typedef struct S_port {
   T_ident        id;
   int            number;
   int            direction;
   int            type;
   struct S_port *next;
} T_port;

/* Port signal node pointer */
typedef T_port *  TP_port;

/*-------------------------------------------------------------------*/

/* prototypes */

void port_clear(void);
void port_add_id(void);
void port_add_id_list(void);
void port_print(TP_port list);
TP_port port_get(void);
```

```
TP_port port_get(int id, TP_port ptr);

/*-------------------------------------------------------------------------*/
#endif
```

*E.22 PORT.CPP*

```
/*
   VHDL PARSER

   File: PORT.C

   Date: 2 July 1992

   This module handles the creation of a port's list of
   input/output signals. Theses functions are called by the BISON
   program.

*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <malloc.h>
#include "thesis.h"
#include "port.h"
#include "misc.h"
/*-------------------------------------------------------------------*/
/* Storage for current port signal list */

static TP_port port_signal_list = NULL;
static int     in_port_count    = 0;
static int     out_port_count   = 0;
/*-------------------------------------------------------------------*/
void port_clear(void)
{
   port_signal_list    = NULL;
   in_port_count       = 0;      // Current port location number
   out_port_count      = 0;      // Current port location number
}

void port_add_id(void)
{
   TP_port new_node;

   if( (new_node = (TP_port)(malloc(sizeof(T_port)))) == NULL ) {
      yyerror("Out of memory in PORT_ADD_ID()");
      exit(124);
   }
   new_node->id      = ident_get();
   switch(new_node->direction) {
   case V_IN:
      new_node->number = in_port_count++;
      break;
   case V_OUT:
      new_node->number = out_port_count++;
      break;
```

```
   default:
      puts("!!!!! Illegal direction in port_add_id, port.cpp");
      exit(125);
   }
   new_node->next    = port_signal_list;
   port_signal_list  = new_node;
}

void port_add_id_list(void)
{
   TP_port        new_node;
   TP_ident_list  new_list;
   TP_ident_list  lst_ptr;

   new_list = lst_ptr = ident_list_get();

   while( lst_ptr != NULL ) {
      if( (new_node = (TP_port)(malloc(sizeof(T_port)))) == NULL ) {
         yyerror("Out of memory in PORT_ADD_ID_LIST()");
         exit(126);
      }
      new_node->id        = lst_ptr->id;
      new_node->direction = direct_get();
      switch(new_node->direction) {
      case V_IN:
         new_node->number = in_port_count++;
         break;
      case V_OUT:
         new_node->number = out_port_count++;
         break;
      default:
         puts("!!!!! Illegal direction in port_add_id_list, port.cpp");
         exit(127);
      }
      new_node->type      = type_get();
      new_node->next      = port_signal_list;
      port_signal_list    = new_node;
      lst_ptr             = lst_ptr->next;
   }
   ident_list_free(new_list);
}

void port_print(TP_port list)
{
   if(list == NULL) {
      puts("Empty port signal list");
   }
   else {
      if(list == CURRENT_LIST ) {
         puts("Found CL; printing current list:");
         list = port_signal_list;
```

```
      }
      while(list != NULL ) {
         printf("Name: ");
         ident_print( &(list->id) );
         printf(" #:%2d",list->number);
         printf(" Dir: ");
         switch(list->direction) {
            case V_IN:  printf("IN ");break;
            case V_OUT: printf("OUT");break;
            default:  printf("Unknown");
         }
         printf(" Type: BIT");
         puts("");
         list = list->next;
      }
   }
}

TP_port port_get(void)
{
   return port_signal_list;
}
/*-------------------------------------------------------------------*/
// Return pointer to port specified by id in supplied port list
// Return NULL if not found
TP_port port_get(int id, TP_port ptr)
{
   while(ptr != NULL && ptr->id != id) {
      ptr = ptr->next;
   }
   if( ptr == NULL ) {
      printf("!!!!! Error - Couldn't find port %d (PORT.CPP L141)",id);
      exit(128);
   }
   return ptr;
}
/*-------------------------------------------------------------------*/
```

*E.23 PORTMAP.H*

```
/*
   VHDL PARSER

   File: PORTMAP.H

   Date: 9 July 1992

   This module handles the creation of a portmap's list of
   input/output signals. Theses functions are called by the BISON
   program.

   Routines:
   ---------
   portmap_clear()       -- Clear current portmap list
   portmap_add_id()      -- Add current ident to current portmap list D
   portmap_add_id_list() -- Add current ident_list to current portmap list
   portmap_print(list)   -- Print supplied portmap list. If CURRENT_LIST,
                            print out the current list.
   portmap_get()         -- Get pointer to current list.
*/
/*-------------------------------------------------------------------*/
#ifndef __portmap_h__
#define __portmap_h__ 1

#ifndef __assoc_h__
#include "assoc.h"
#endif


/* portmap signal node */
typedef struct S_portmap {
   TP_assoc       assoc_list;
   struct S_portmap *next;
} T_portmap;

/* portmap signal node pointer */
typedef T_portmap *  TP_portmap;

/*-------------------------------------------------------------------*/

/* prototypes */
/*-------------------------------------------------------------------*/
void portmap_list_clear(void);
void portmap_list_free(TP_portmap list);
void portmap_list_add_node(TP_assoc list);
void portmap_list_print(TP_portmap list);
TP_portmap portmap_get(void);

/*-------------------------------------------------------------------*/
#endif
```

## *E.24 PORTMAP.CPP*

```
/*
   VHDL PARSER

   File: PORTMAP.C

   Date: 9 July 1992

   This module handles the creation of a portmap's list of
   input/output signals. Theses functions are called by the BISON
   program.

*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <malloc.h>
#include "thesis.h"
#include "portmap.h"
#include "misc.h"
/*-------------------------------------------------------------------*/
/* Storage for current portmap signal list */

static TP_portmap portmap_list = NULL;

/*-------------------------------------------------------------------*/
void portmap_list_clear(void)
{
   portmap_list = NULL;
}

void portmap_list_free(TP_portmap list)
{
   TP_portmap ptr;

   if(list == CURRENT_LIST) {
      list = portmap_list;
   }
   while(list != NULL) {
      ptr = list->next;
      free(list);
      list = ptr;
   }
}

void portmap_list_add_node(TP_assoc list)
{
   TP_portmap new_node;

   if( (new_node = (TP_portmap)(malloc(sizeof(T_portmap)))) == NULL ) {
```

```
      yyerror("Out of memory in portmap_ADD_ID()");
      exit(107);
   }
   new_node->assoc_list = list;
   new_node->next       = portmap_list;
   portmap_list  = new_node;
}

void portmap_list_print(TP_portmap list)
{
   if(list == NULL) {
      puts("Empty portmapiation list");
   }
   else {
      if(list == CURRENT_LIST ) {
         puts("Found CL; printing current portmap list:");
         list = portmap_list;
      }
      while(list != NULL ) {
         puts("Association list:");
         assoc_list_print( list->assoc_list);
         list = list->next;
      }
   }
}

TP_portmap portmap_get(void)
{
   return portmap_list;
}
/*-------------------------------------------------------------------------*/
```

## *E.25 PROCESS.H*

```
/*
   VHDL PARSER

   File: PROCESS.H

   Date: 14 July 1992

   This module handles the creation of an process by the
   VHDL parser. These routines are called by the BISON program.

   Routines:
   ---------
process_clear() -- Clear current process settings
process_add_mcode() - Add current mcode
   process_print()    - Print out specified process - NULL prints
       out current process
*/

/*-----------------------------------------------------------------*/
#ifndef __process_h__
#define __process_h__ 1

#ifndef __mcode_h__
#include "mcode.h"
#endif

typedef struct S_process {
   TP_mcode  mcode;
} T_process;

#ifdef __process_c__
T_process NULL_PROCESS ={NULL};
#else
extern T_process NULL_PROCESS;
#endif

/* prototypes */

void        process_clear(void);
void        process_add_mcode(void);
T_process   process_get(void);
void        process_c_print(void);
void        process_print( T_process *ent);
int         create_c_sim_process(void);
int         create_sim_process(T_process *proc);
void        new_sim_block(int arch_id);
void        add_code_to_sim_block(int code_id, char *code_title);
void        finish_sim_block(void);
void        create_all_hypo(void);
```

```
/*-------------------------------------------------------------------*/
#endif
```

## E.26 PROCESS.CPP

```
/*
   VHDL PARSER

   File: PROCESS.C

   Date: 14 July 1992

   This module handles the creation of an process by the
   VHDL parser. These routines are called by the BISON program.
   Fault hypotheses are also added in this module.

*/
#define __process_c__ 1

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include "thesis.h"
#include "process.h"
#include "block.hpp"
#include "code.hpp"
/*-------------------------------------------------------------------*/
/* Current process being created */

static T_process current_process;

/*-------------------------------------------------------------------*/
void process_clear(void)
{
   current_process.mcode = NULL;
}
/*-------------------------------------------------------------------*/
void process_add_mcode(void)
{
   current_process.mcode = mcode_get();
}
/*-------------------------------------------------------------------*/
T_process process_get(void)
{
   return current_process;
}
/*-------------------------------------------------------------------*/
void process_c_print(void)
{
   process_print(&current_process);
}
/*-------------------------------------------------------------------*/
void process_print( T_process *ent)
{
```

```
   if( ent->mcode == NULL ) {
      puts("Null Process");
   }
   else {
      puts("process mcode list");
      mcode_print( ent->mcode);
   }
}
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
// Create code-structure for simulator use. This routine will take the
// supplied code process and create a CODE block. This routine will be
// called by create_sim_block()

int create_c_sim_process(void)
{
   return create_sim_process(&current_process);
}

int create_sim_process(T_process *proc)
{
   static int code_count = 0; // ID for new code block; return value

   Code code_block;

   code_block         = Code(code_count);
   T_mcode *mcode_ptr = proc->mcode;

   while(mcode_ptr != NULL ) {
      code_block.add_mcode(MCode(mcode_ptr->code));
      mcode_ptr = mcode_ptr->next;
   }
   add_code_block(code_block);

   return code_count++;
}
/*-------------------------------------------------------------------*/
// Storage for new sim block object

static block new_block;

void new_sim_block(int arch_id)
{
   new_block = block(arch_id);
}

void add_code_to_sim_block(int code_id, char *code_title)
{
   new_block.add_code(code_id, code_title);
}
```

```
void finish_sim_block(void)
{
   add_block_inst(new_block);
}
/*-------------------------------------------------------------------*/

int hypo_in_stuck_hi( Code &good_code, int get_number)
{
   int  mcode_pos   = 0;
   int  code_get_no = M_NULL;
   int  found_get   = FALSE;
   char title_str[MAX_STR_LEN+1];
   MCode mcode_ref;

   mcode_clear_list();
   process_clear();
   while( mcode_pos < good_code.get_code_blk_len()) {
      mcode_ref = good_code.get_mcode_at(mcode_pos);
      if( mcode_ref.get_op_code() == M_GET ) {
         if(code_get_no == get_number) {
            mcode_add(M_POP);  // Remove signal offset
            mcode_add(1);      // Stuck HI
            found_get = TRUE;
         }
         else {
            mcode_add(mcode_ref.get_op_code());
         }
      }
      else {
         mcode_add(mcode_ref.get_op_code());
      }
      code_get_no = mcode_ref.get_op_code();
      mcode_pos++;
   }
   if(found_get) {
      process_add_mcode();
      sprintf(title_str, "Input #%d stuck hi ",get_number);
      add_code_to_sim_block(create_c_sim_process(), title_str);
   }
   return found_get;
}

/*-------------------------------------------------------------------*/
int hypo_in_stuck_low( Code &good_code, int get_number)
{
   int  mcode_pos   = 0;
   int  code_get_no = M_NULL;
   int  found_get   = FALSE;
   char title_str[MAX_STR_LEN+1];
   MCode mcode_ref;
```

```
   mcode_clear_list();
   process_clear();
   while( mcode_pos < good_code.get_code_blk_len()) {
      mcode_ref = good_code.get_mcode_at(mcode_pos);
      if( mcode_ref.get_op_code() == M_GET ) {
         if(code_get_no == get_number) {
            mcode_add(M_POP);  // Remove signal offset
            mcode_add(0);      // Stuck LOW
            found_get = TRUE;
         }
         else {
            mcode_add(mcode_ref.get_op_code());
         }
      }
      else {
         mcode_add(mcode_ref.get_op_code());
      }
      code_get_no = mcode_ref.get_op_code();
      mcode_pos++;
   }
   if(found_get) {
      process_add_mcode();
      sprintf(title_str, "Input #%d stuck low",get_number);
      add_code_to_sim_block(create_c_sim_process(), title_str);
   }
   return found_get;
}

/*-------------------------------------------------------------------*/
int hypo_out_stuck_hi( Code &good_code, int post_number)
{
   int  mcode_pos     = 0;
   int  found_post_no = 0;
   int  found_post    = FALSE;
   char title_str[MAX_STR_LEN+1];
   MCode mcode_ref;

   mcode_clear_list();
   process_clear();
   while( mcode_pos < good_code.get_code_blk_len()) {
      mcode_ref = good_code.get_mcode_at(mcode_pos);
      if( mcode_ref.get_op_code() == M_POST ) {
         if(found_post_no == post_number) {
            mcode_add(0);       // Temp store addr to save time
            mcode_add(M_STORE); // Store delta time
            mcode_add(M_POP);   // Remove old value
            mcode_add(1);       // Stuck HI
            mcode_add(0);       // Temp store addr
            mcode_add(M_RETRV); // Get stored delta time
            mcode_add(M_POST);  // POST
            found_post = TRUE;
```

```
         }
         else {
            mcode_add(mcode_ref.get_op_code());
         }
         found_post_no++;
      }
      else {
         mcode_add(mcode_ref.get_op_code());
      }
      mcode_pos++;
   }
   if(found_post) {
      process_add_mcode();
      sprintf(title_str, "Outut #%d stuck hi ",post_number);
      add_code_to_sim_block(create_c_sim_process(), title_str);
   }
   return found_post;
}

/*-------------------------------------------------------------------*/
int hypo_out_stuck_low( Code &good_code, int post_number)
{
   int  mcode_pos     = 0;
   int  found_post_no = 0;
   int  found_post    = FALSE;
   char title_str[MAX_STR_LEN+1];
   MCode mcode_ref;

   mcode_clear_list();
   process_clear();
   while( mcode_pos < good_code.get_code_blk_len()) {
      mcode_ref = good_code.get_mcode_at(mcode_pos);
      if( mcode_ref.get_op_code() == M_POST ) {
         if(found_post_no == post_number) {
            mcode_add(0);       // Temp store addr to save time
            mcode_add(M_STORE); // Store delta time
            mcode_add(M_POP);   // Remove old value
            mcode_add(0);       // Stuck LOW
            mcode_add(0);       // Temp store addr
            mcode_add(M_RETRV); // Get stored delta time
            mcode_add(M_POST);  // POST
            found_post = TRUE;
         }
         else {
            mcode_add(mcode_ref.get_op_code());
         }
         found_post_no++;
      }
      else {
         mcode_add(mcode_ref.get_op_code());
      }
```

```
      mcode_pos++;
   }
   if(found_post) {
      process_add_mcode();
      sprintf(title_str, "Outut #%d stuck low",post_number);
      add_code_to_sim_block(create_c_sim_process(), title_str);
   }
   return found_post;
}

/*-------------------------------------------------------------------*/

// The working model of a process is defined as code #0
#define WORKING_MODEL 0

void create_all_hypo(void)
{
   int get_no,
       post_no;

   Code  &code_ptr  = get_code_block(new_block.get_code(WORKING_MODEL));

   // Hypothesize inputs of component stuck high
   post_no = 0;
   while( hypo_out_stuck_hi (code_ptr, post_no++)) {}

   // Hypothesize inputs of component stuck low
   post_no = 0;
   while( hypo_out_stuck_low(code_ptr, post_no++)) {}

   // Hypothesize inputs of component stuck high
   get_no = 0;
   while( hypo_in_stuck_hi (code_ptr, get_no++)) {}

   // Hypothesize inputs of component stuck low
   get_no = 0;
   while( hypo_in_stuck_low(code_ptr, get_no++)) {}

}
/*-------------------------------------------------------------------*/
```

## E.27 SIGNAL.H

```
/*
   VHDL PARSER

   File: SIGNAL.H

   Date: 7 July 1992

   Routines:
   ---------
   signal_clear_list()  -- Clear current signal list pointer
   signal_add_id()      -- Add current id to current signal list !!!NOT USED
   signal_add_id_list() -- Add current id_list to current signal list
   signal_print(list)   -- Print specified signal list. If CURRENT_LIST,
                           print current signal list.
   signal_get()         -- Get pointer to current signal list.
*/
/*-------------------------------------------------------------------*/
#ifndef __signal_h__
#define __signal_h__ 1

#ifndef __ident_h__
#include "ident.h"
#endif


/* Port signal node */
typedef struct S_signal {
   T_ident        id;
   int            type;
   struct S_signal *next;
} T_signal;

/* Port signal node pointer */
typedef T_signal *  TP_signal;

/*-------------------------------------------------------------------*/

/* prototypes */
#ifdef __BORLANDC__
void     signal_clear_list(void);
void     signal_add_id(void);
void     signal_add_id_list(void);
void     signal_print(TP_signal list);
TP_signal  signal_get(void);
#else
void     signal_clear_list();
void     signal_add_id();
void     signal_add_id_list();
void     signal_print();
TP_signal  signal_get();
```

```
#endif

/*-----------------------------------------------------------------------*/
#endif
```

*E.28 SIGNALP.CPP*

```
/*
   VHDL PARSER

   File: SIGNAL.C

   Date: 7 July 1992

   This module handles the creation of signals.
*/

#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "thesis.h"
#include "signal.h"
#include "misc.h"
/*-------------------------------------------------------------------*/
/* Storage for current signal signal list */

static TP_signal signal_signal_list = NULL;

/*-------------------------------------------------------------------*/
void signal_clear_list(void)
{
   signal_signal_list = NULL;
}

void signal_add_id(void)
{
   TP_signal new_node;

   if( (new_node = (TP_signal)(malloc(sizeof(T_signal)))) == NULL ) {
      yyerror("Out of memory in PORT_ADD_ID()");
      exit(108);
   }
   new_node->id      = ident_get();
   new_node->next    = signal_signal_list;
   signal_signal_list  = new_node;
}

void signal_add_id_list(void)
{
   TP_signal        new_node;
   TP_ident_list  new_list;
   TP_ident_list  lst_ptr;

   new_list = lst_ptr = ident_list_get();

   while( lst_ptr != NULL ) {
      if( (new_node = (TP_signal)(malloc(sizeof(T_signal)))) == NULL ) {
```

```
            yyerror("Out of memory in SIGNAL_ADD_ID_LIST()");
            exit(109);
         }
         new_node->id        = lst_ptr->id;
         new_node->type      = type_get();
         new_node->next      = signal_signal_list;
         signal_signal_list  = new_node;
         lst_ptr           = lst_ptr->next;
      }
      ident_list_free(new_list);
}

void signal_print(TP_signal list)
{
      if(list == NULL) {
         puts("Empty signal signal list");
      }
      else {
         if(list == CURRENT_LIST ) {
            puts("Found NULL; printing current signal list:");
            list = signal_signal_list;
         }
         while(list != NULL ) {
            printf("Name: ");
            ident_print( &(list->id) );
            printf(" Type: BIT");
            puts("");
            list = list->next;
         }
      }
}

TP_signal signal_get(void)
{
      return signal_signal_list;
}
/*-----------------------------------------------------------------*/
```

# Appendix F. *Simulator/Diagnostic Source Code*

## *F.1 Overview*

This appendix contains the code for Calvin's VHDL simulator and diagnostic routines. A module generally consists of a header file and associated code file. These have the same or similar names, with a ".hpp" extension for the header and a ".cpp" extension for the code file. The following modules were discussed in Chapter III:

- **BEHAVE** - section 3.3.4.6
- **BLOCK** - section 3.3.4.7
- **CODE** - section 3.3.4.8
- **MCODE** - section 3.3.4.9
- **SIGNAL** - section 3.3.4.5.

The following modules handle specific minor tasks:

- **COMSEN** - manage the lists of commands and sensors
- **INT** - A Object shell for an integer. This module was required so that integer values could be used with Borland's container library.
- **STAT** - This module collected the statistics discussed in section 4.1.1.
- **THESIS** - This module contained the various constants used throughout the simulator/diagnostic routines.

The module **MAIN** contains Calvin's user interface. Here is were Calvin parses the command line, and where Calvin's system flags are set. Calvin opens the source file, and sends it to the VHDL parser. Calvin then calls the diagnostic routines contained in the module named **CALVIN**.

The **VHDL** module is Calvin's VHDL simulator. Most of the VHDL simulation routines are gathered here. This module uses the **signal**, **behave**, **block**, and **code** objects to perform the simulation. The simulation is controlled by a function at the end of the **VHDL** module, **vhdl_main_loop()** . The activation record used by the VHDL simulator is defined in the module **AR**.

The diagnostic routines are gathered into the module **CALVIN**. These routines include suspect collection and fault insertion. The main diagnostic algorithm is also in **CALVIN**.

### *F.2 AR.HPP*

```
//
// AR.HPP
//
// Activation Record Class
//
// Modified 16 July 1992
//
#ifndef __AR_HPP__
#define __AR_HPP__ 1

#include <sortable.h>

#define ActiveRecordClass 222

class ActiveRecord : public Sortable {
public:
   ActiveRecord() {
      time     = -1;
      sr_ptr   = -1;
      value    = -1;
   }
   ActiveRecord(int new_time, int new_sr_ptr, int new_value) {
      time     = new_time;
      sr_ptr   = new_sr_ptr;
      value    = new_value;
   }
   int get_sr_ptr(void) {
      return sr_ptr;
   }
   int get_value(void) {
      return value;
   }
```

```
   int get_time(void) {
      return time;
   }

   virtual int isEqual( const Object& otherObj ) const {
      return time == ((ActiveRecord&) otherObj).time;
   }
   virtual int isLessThan( const Object& otherObj ) const {
      return time < ((ActiveRecord &) otherObj).time;
   }
   virtual classType isA() const {return ActiveRecordClass;}

   virtual char *nameOf() const {return "Active Record";}
   virtual hashValueType hashValue() const {
      return time;
   }
   virtual void printOn( ostream& coutt ) const {
      coutt << "Time: " << time << " srptr: " << sr_ptr
            << " Value:" << value;
   }

private:
   int  time;
   int   sr_ptr;
   int   value;
};
#endif
```

## *F.3 BEHAVE.HPP*

```
//
//
// BEHAVE.HPP
//
// Behave Class
//
// 16 July 1992
//
// Behavie instance object
//
//----------------------

#include "thesis.h"
#define MAX_CODE 20 // Maximum number of bodies for behave

#define MAX_BEHAVE_INST 40 // Max number of behaviors in simulation

class Behave {
public:
   Behave();
   Behave(int new_id);
   int get_id(void);              // Get id number

   void set_code_select(int);     // Select code for execution
   int  get_current_select(void);// Get current code nuber
   void set_block_id(int);        // Set block id to exec for this behave
   int  get_block_id(void);       // Get block id to exec
   int  get_code_count(void);     // Return number of bodies

   void add_input(int);           // Add new input to input list
   void add_input(int,int);       // Add new input to position in list
   int  get_input(int);           // Get input id
   int  get_input_count(void);    // Return number of inputs

   void add_output(int);          // Add new output to output list
   void add_output(int,int);      // Add new output to position in list
   int  get_output(int);          // Get output id
   int  get_output_count(void);   // Return number of outputs

   void print(char *);            // Debug print

private:
   int   id;               // Int name of behave
   int   block_id;         // ID of block to exec for this behave
   int   code_select;      // Current code for Behave execution
   int   input[MAX_IN];    // List of inputs to Behave
   int   last_in;          // Last input added
   int   output[MAX_OUT];  // List of outputs from Behave
   int   last_out;         // Last output added
};
```

```
//-----------------------------------------------------------------------
// Behave instance storage management routines

void     reset_behave_inst(void);
void     add_behave_inst( Behave &new_behave);
int      get_last_behave_inst(void);
int      get_behave_id_at(int pos); // Get behave it at position
Behave  &get_behave_inst(int id);
void     behave_inst_print(char *s);

void     flush_behave_mark(void);
void     mark_behave(int id);
int      is_behave_marked(int pos);

//-----------------------------------------------------------------------
```

## F.4 BEHAVE.CPP

```
//
// BEHAVE.CPP
//
// Behave Class
//
// 16 July 1992
//----------------------
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <assert.h>

#include "behave.hpp"
#include "block.hpp"
//-------------------------------------------------------------------
// Storage for Behave instance

static Behave behave_storage[MAX_BEHAVE_INST];
static int    behave_mark[MAX_BEHAVE_INST];
static int  last_behave_inst;

//-------------------------------------------------------------------
// Local function prototypes
int translate_id_to_pos(int id);
//-------------------------------------------------------------------
// Clear behave instance mark list
void flush_behave_mark(void)
{
   for(int i=0;i<MAX_BEHAVE_INST;++i) {
      behave_mark[i] = FALSE;
   }
}
//-------------------------------------------------------------------
// Mark a behave instance
void mark_behave(int id)
{
   behave_mark[translate_id_to_pos(id)] = TRUE;
}
//-------------------------------------------------------------------
// Return mark status of a behave
int is_behave_marked(int id)
{
   return behave_mark[translate_id_to_pos(id)];
}
//-------------------------------------------------------------------
//-------------------------------------------------------------------
void reset_behave_inst(void)
{
   last_behave_inst = 0;
}
```

```
//-----------------------------------------------------------------
void add_behave_inst( Behave &new_behave)
{
   assert(last_behave_inst<MAX_BEHAVE_INST);
   behave_storage[last_behave_inst++] = new_behave;
}
//-----------------------------------------------------------------
int translate_id_to_pos(int id)
{
   for(int i=0; i<last_behave_inst; ++i) {
      if( behave_storage[i].get_id() == id ) {
         return i;
      }
   }
   printf("Invalid id numbr #%d in translate_id_to_pos()\n",id);
   exit(1);
   return -1;
}
//-----------------------------------------------------------------
Behave &get_behave_inst(int id)
{
   for(int i=0; i<last_behave_inst; ++i) {
      if( behave_storage[i].get_id() == id ) {
         return behave_storage[i];
      }
   }
   printf("!!!!!! Bad id '%d'in get_behave_inst---behave.cpp\n",id);
   exit(1);
   return behave_storage[0];
}
//-----------------------------------------------------------------
int      get_behave_id_at(int pos)
{
   return behave_storage[pos].get_id();
}
//-----------------------------------------------------------------
int  get_last_behave_inst(void)
{
   return last_behave_inst;
}
//-----------------------------------------------------------------
void behave_inst_print(char *s)
{
   printf(s);
   ıor(int i=0; i<last_behave_inst; ++i) {
      printf("-----behave inst  %2d-----\n",i);
      behave_storage[i].print("");
      getch();
   }
   puts("----------------------");
}
```

```
//=====================================================================
Behave::Behave()
{
   id = 0;
   last_in = last_out = 0;
   code_select  = 0;
}
//---------------------------------------------------------------------
Behave::Behave(int new_id)
{
   id = new_id;
   last_in = last_out = 0;
   code_select  = 0;
}
//---------------------------------------------------------------------
int Behave::get_id(void)
{
   return id;
}
//---------------------------------------------------------------------
void Behave::set_code_select(int new_code_select)
{
   code_select = new_code_select;
}
//---------------------------------------------------------------------
int  Behave::get_current_select(void)
{
   return code_select;
}
//---------------------------------------------------------------------
void Behave::set_block_id(int new_block_id)
{
   block_id = new_block_id;
}
//---------------------------------------------------------------------
int  Behave::get_block_id(void)
{
   return block_id;
}
//---------------------------------------------------------------------
int  Behave::get_code_count(void)
{
   return get_block_inst(block_id).get_code_count();
}
//---------------------------------------------------------------------
void Behave::add_input(int input_id)
{
   assert(last_in < MAX_IN);

   input[last_in++] = input_id;
}
```

```
//-----------------------------------------------------------------
void Behave::add_input(int input_id, int input_pos)
{
   assert(input_pos < MAX_IN);

   input[input_pos++] = input_id;
   if(input_pos>=last_in) {
      last_in = input_pos;
   }
}
//-----------------------------------------------------------------
int  Behave::get_input(int input_no)
{
   assert(input_no<last_in);
   assert(input_no>=0);

   return input[input_no];
}
//-----------------------------------------------------------------
int  Behave::get_input_count(void)
{
   return last_in;
}
//-----------------------------------------------------------------
void Behave::add_output(int output_id)
{
   assert(last_out < MAX_OUT);

   output[last_out++] = output_id;
}
//-----------------------------------------------------------------
void Behave::add_output(int output_id, int output_pos)
{
   assert(output_pos < MAX_OUT);

   output[output_pos++] = output_id;
   if(output_pos>=last_out) {
      last_out = output_pos;
   }
}
//-----------------------------------------------------------------
int  Behave::get_output(int output_no)
{
   if(output_no >= last_out) {
      printf("ERR: this=%d output_no=%d last_out=%d\n", id,output_no,last_out);
      exit(-1);
   }
   assert(output_no<last_out);
   assert(output_no>=0);

   return output[output_no];
```

```
}
//-----------------------------------------------------------------
int  Behave::get_output_count(void)
{
   return last_out;
}
//-----------------------------------------------------------------
void Behave::print(char *s)
{
   printf(s);
   printf("Behave id: %2d  Block id:%2d Current code: %2d\n",id,
      block_id,code_select);
   puts("Inputs:");
   for(int i=0;i<last_in;++i) {
      printf("%2d| input  id %2d\n",i,input[i]);
   }
   puts("Outputs:");
   for(i=0;i<last_out;++i) {
      printf("%2d| output id %2d\n",i,output[i]);
   }
}
//-----------------------------------------------------------------
```

*F.5 BLOCK.HPP*

```
//
//
// BLOCK.HPP
//
// Process Block Class
//
// 30 July 1992
//----------------------
#ifndef _BLOCK_HPP_
#define _BLOCK_HPP_

#define MAX_CODES 20       // Maximum number of code bodies for block
#define MAX_BLOCK_INST 10  // Max number of blocks in simulation
#define MAX_STR_LEN 20
//-------------------------------------------------------------------
class block {
public:
   block(int id);                 // Constructor
   block(void);
   ~block();                      // Destructor
   int  get_id(void);             // Get id number
   void add_code(int,char *);     // Add new code to code list
   int  get_code(int);            // Get code id
   int  get_code_count(void);     // Return number of bodies
   char *hyp_str_get(int);        // Return hypothesis title
   void print(char *s);           // Print block description
private:
   int   id;                      // Int name of block
   int   sim_code_id[MAX_CODES];  // VHDL microcode code number
   char  hyp_str[MAX_CODES][MAX_STR_LEN+1];  // Strings for hypothesis' name
   int   last_code_no;            // Last code number added
};
//-------------------------------------------------------------------
// block instance storage management routines

void     reset_block_inst(void);
void     add_block_inst( block &new_block);
int      get_last_block_inst(void);
int      get_block_id_at(int pos); // Get block it at position
block   &get_block_inst(int id);
void     block_inst_print(char *s);
//-------------------------------------------------------------------
#endif
```

## F.6 BLOCK.CPP

```
//
//
// BLOCK.CPP
//
// Process Block Class
//
// 30 July 1992
//----------------------

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>
#include <assert.h>
#include "block.hpp"
//---------------------------------------------------------------------
// Storage for block instance

static block block_storage[MAX_BLOCK_INST];
static int   last_block_inst;

//---------------------------------------------------------------------
block::block(int new_id)
{
   id           = new_id;
   last_code_no = 0;
   // Clear all hypothesis strings
   for(int i=0;i<MAX_CODES;++i) {
      strcpy(hyp_str[i], "---");
   }
}
block::block(void)
{
   id           = -1;
   last_code_no = 0;
   // Clear all hypothesis strings
   for(int i=0;i<MAX_CODES;++i) {
      strcpy(hyp_str[i], "---");
   }
}
block::~block()
{
//   printf("BLoCK::~DELETE  %p\n",this);getch();
}
//---------------------------------------------------------------------
int block::get_id(void)
{
   return id;
}
```

```
//-----------------------------------------------------------------
void block::add_code(int code_id, char *new_hyp_str)
{
   assert(last_code_no <MAX_CODES);

   sim_code_id[last_code_no] = code_id;
   strncpy(hyp_str[last_code_no], new_hyp_str, MAX_STR_LEN);
   last_code_no++;
}
//-----------------------------------------------------------------
char *block::hyp_str_get(int number)
{
   return hyp_str[number];
}
//-----------------------------------------------------------------
int  block::get_code(int code_no)
{
   assert(code_no<last_code_no);
   assert(code_no>=0);

   return sim_code_id[code_no];
}
//-----------------------------------------------------------------
int  block::get_code_count(void)
{
   return last_code_no;
}
//-----------------------------------------------------------------
void block::print(char *s)
{
   printf(s);
   printf("Block id: %2d\n",id);
   puts("Codes:");
   for(int i=0;i<last_code_no;++i) {
      printf("%2d| code   id %2d\n",i,sim_code_id[i]);
   }
}
//-----------------------------------------------------------------
//-----------------------------------------------------------------
void reset_block_inst(void)
{
   last_block_inst = 0;
}
//-----------------------------------------------------------------
void add_block_inst( block &new_block)
{
   assert(last_block_inst<MAX_BLOCK_INST);
   block_storage[last_block_inst++] = new_block;
}
//-----------------------------------------------------------------
block &get_block_inst(int id)
```

```
{
   for(int i=0; i<last_block_inst; ++i) {
      if( block_storage[i].get_id() == id ) {
         return block_storage[i];
      }
   }
   printf("!!!!!! Bad id |%2d| in get_block_inst---block.cpp\n",id);
   exit(116);
   return block_storage[0];
}
//---------------------------------------------------------------------
int  get_block_id_at(int pos)
{
   return block_storage[pos].get_id();
}
//---------------------------------------------------------------------
int  get_last_block_inst(void)
{
   return last_block_inst;
}
//---------------------------------------------------------------------
void block_inst_print(char *s)
{
   printf(s);
   for(int i=0; i<last_block_inst; ++i) {
      printf("-----block inst  %2d-----\n",i);
      block_storage[i].print("");
      getch();
   }
   puts("----------------------");
}
//=====================================================================
//---------------------------------------------------------------------
//---------------------------------------------------------------------
```

## F.7 CALVIN.CPP

```
//
//
// CALVIN.CPP
//
// 16 Jul 92
//
// This module is where most of the modules that make up the current
// configuration of Calvin. During developement, routines were created
// using this module. As they were completed, they were spawned off into
// their own modules. The current state of Calvin developement is in this
// module.
//
//
//
// void print_bi_queue()           -- Debuge routine
// int  get_bi_from_signal()  -- Get the Behave that drives the signal
// void init_suspect_queue()  -- init queue
// void collect_bi_suspects() -- Original depth-1st collection routine
// void sus_depth_1st(int sr_id)          --+
// void collect_bi_suspects_2( int sr_id ) --+- Modified collection routine.
//                Break when encounter a Behave that is already in the queue.
//
// These routines make up the suspect collection part of Calvin.
//
//
// void sim_signal_init(void) -- init routine
// void sim_set_up()          -- init routine
// int  sensor_comp()         -- Compare simulated sensors with outside sensors
// void load_out_val()        -- Get outside sensor values
// void faultify_behave()     -- "Break" the circuit.
// void diagnose()            -- The Diagnose algorithm
// void run_exam()            -- Run Calvin
//-------------------------------------------------------------------------
//
#include <stdio.h>
#include <conio.h>
#include <assert.h>
#include <string.h>
#include <stdlib.h>

#include <queue.h>

#include "thesis.h"
#include "signal.hpp"
#include "behave.hpp"
#include "block.hpp"
#include "code.hpp"

#include "int.hpp"
#include "vhdl.hpp"
```

```
#include "comsen.hpp"
#include "stat.hpp"
//-------------------------------------------------------------------
Queue bi_queue;  // Queue of Behave Instance suspects
//-------------------------------------------------------------------
// Debug function to print suspect BI queue
void print_bi_queue(void)
{
   Queue temp_q = bi_queue;

   puts("Queue of Behavioral Instance suspects");
   while(!temp_q.isEmpty()) {
      cout << temp_q.get() << endl;
   }
   puts("-----------------------------------");
   exit(0);
}
//-------------------------------------------------------------------
// Get BI id# of instance that drives specified signal
int get_bi_from_signal(int sr_id)
{
   SignalRecord &sr_ptr = get_signal_rec(sr_id);
   return sr_ptr.get_driver_bi();
}
//-------------------------------------------------------------------
// Flush out the suspect queues
void init_suspect_queue(void)
{
   bi_queue.flush();
}
//-------------------------------------------------------------------
// Collect list of suspects from a possible discrepant signal
// Uses depth first approach

//int level = 0;

void collect_bi_suspects( int sr_id)
{

   Integer *suspect_bi;

   suspect_bi = new Integer(get_bi_from_signal(sr_id));
   // Make sure we're not at the top (command level)
   if(suspect_bi->value() != COMMAND_SR) {
      bi_queue.put( *suspect_bi);

      // Recurse upstream from behave instance
      Behave &bi_ptr = get_behave_inst(suspect_bi->value());
      for( int i=0; i<bi_ptr.get_input_count();++i) {
         collect_bi_suspects(bi_ptr.get_input(i));
         inc_stat(NO_SUSPECTS);
```

```
      }
   }
   else {
//      puts("Reached command level");
   }
}

//-----------------------------------------------------------------

void sus_depth_1st(int sr_id)
{
   Integer *suspect_bi;

   suspect_bi = new Integer(get_bi_from_signal(sr_id));
   // Make sure we're not at the top (command level)
   if(suspect_bi->value() != COMMAND_SR) {
      // Make sure suspect not in queue already
      if( !is_behave_marked(suspect_bi->value()) ) {
         mark_behave(suspect_bi->value());
         bi_queue.put( *suspect_bi);

         // Recurse upstream from behave instance
         Behave &bi_ptr = get_behave_inst(suspect_bi->value());
         for( int i=0; i<bi_ptr.get_input_count();++i) {
            sus_depth_1st(bi_ptr.get_input(i));
            inc_stat(NO_SUSPECTS);
         }
      }
   }
   else {
//      puts("Reached command level");
   }
}



void collect_bi_suspects_2( int sr_id )
{
   flush_behave_mark();
   init_suspect_queue();
   sus_depth_1st(sr_id);
}



//-----------------------------------------------------------------
void sim_signal_init(void)
{
   process_init();
   vhdl_main_loop();
}
```

```
//-----------------------------------------------------------------
void sim_set_up(
   int  *in_val,
   int   in_val_last )
{
   for(int i=0;i<in_val_last;++i) {
      post_signal(0,get_command(i),*(in_val+i));
   }
}
//-----------------------------------------------------------------
// Compare simulated values (out_val) with recorded (rec_val)
// Return TRUE if same, FALSE if not
int sensor_comp(
   int   *out_val,
   int   *rec_val,
   int    out_val_last )
{
   for(int i=0;i<out_val_last;++i) {
      if(out_val[i] != rec_val[i]) {
         return FALSE;
      }
   }
   return TRUE;
}
//-----------------------------------------------------------------
void load_out_val(
   int  *out_val,
   int   out_val_last )
{
   for(int i=0;i<out_val_last;++i) {
      SignalRecord si_ptr = get_signal_rec(get_sensor(i));
      *(out_val+i) = si_ptr.get_cval();
   }
}
//-----------------------------------------------------------------
void faultify_behave(
   int   bi_id,
   int  *in_val,
   int   in_val_last,
   int  *out_val,
   int   out_val_last,
   int  *rec_val )
{
   Behave &bi_ptr = get_behave_inst(bi_id);
   if( is_flag_set(PRINT_HYPO)) {
      printf("For %2d, we have %d bodies.\n",
         bi_id, bi_ptr.get_code_count());
   }
   for(int i=1;i<bi_ptr.get_code_count(); ++i) {
      if(is_flag_set(PRINT_HYPO)) {
         printf("Selecting fault condition #%d\n",i);
```

```
      }
      // Hypothesize error
      bi_ptr.set_code_select(i);
      inc_stat(NO_HYPO_CHECKED);
      // Re-simulate
      sim_signal_init();
      sim_set_up(in_val,in_val_last);
      vhdl_main_loop();
      load_out_val(out_val,out_val_last);
      // Compare outputs
      if(sensor_comp(out_val,rec_val,out_val_last)) {
         if(is_flag_set(PRINT_TRIV)) {
            printf(">>>>>> Found a Suspect: %s at %d <<<<<<\n",
               (get_block_inst( bi_ptr.get_block_id() ).hyp_str_get(i) ),
               bi_id );
         }
         else {
            printf("%s at %d: Suspect\n",
               (get_block_inst( bi_ptr.get_block_id() ).hyp_str_get(i) ),
               bi_id );
         }
         inc_stat(NO_FAULTS_FOUND);
      }
      else {
         if(is_flag_set(PRINT_HYPO)) {
            printf("Ruled out hypothesis %d\n",i);
         }
      }
      // Clear fault
      bi_ptr.set_code_select(0);
   }
}
//-----------------------------------------------------------------
void diagnose(
   int  *in_val,
   int   in_val_last,
   int  *out_val,
   int   out_val_last,
   int  *rec_val )
{
   int i;
   int found_problem = FALSE;

   // Set up commands
   sim_set_up(in_val,in_val_last);
   vhdl_main_loop();


   // Get simulated values
   load_out_val(out_val,out_val_last);
```

```
    if(is_flag_set(PRINT_1SIM)) {
       puts("Correct operation results:");
       for(int j=0;j<out_val_last;++j) {
          printf("%2d:  Signal #%2d = %2d\n",j,
             get_sensor(j),out_val[j]);
       }
    }

    // Check for problems
    for(i=0;i<out_val_last;++i) {
       if(out_val[i]!=rec_val[i]) {
          found_problem = TRUE;
          if(is_flag_set(PRINT_TRIV)) {
             printf("We have a problem at sensor #%2d (Signal %03d):%2d != %2d\n",
                i,get_sensor(i), out_val[i],rec_val[i]);
          }
          else {
             printf("###%3d###\n",get_sensor(i));
          }

          if(is_flag_set(COLLECT__2) ){
             collect_bi_suspects_2(get_sensor(i));
          }
          else {
             collect_bi_suspects(get_sensor(i));
          }
          if(is_flag_set(PRINT_SUSP) ) {
             print_bi_queue();
          }
          while( !bi_queue.isEmpty() ) {
             faultify_behave(
                ((Integer &)bi_queue.get()).value(),
                in_val,in_val_last,
                out_val,out_val_last,rec_val );
          }
          if( is_flag_set(INSERT_BRK) ) {
             break;
          }
       }
    }

    // If no problems found, state so
    if(!found_problem) {
       puts("No problems found");
    }
}
//---------------------------------------------------------------------
void run_exam(void)
{
    int   in_val [MAX_COMMANDS],
          out_val[MAX_SENSORS],
```

```
        rec_val[MAX_SENSORS];
    int   i;
    int   in_last  = get_last_command();
    int   out_last = get_last_sensor();

    // Get circuit to steady state
    sim_signal_init();

    if(G_con_flag) {
       printf("Enter values for the %d command signals:\n",in_last);
       for(i=0;i<in_last;++i) {
          printf("Command   %s:",get_signal_rec(get_command(i)).get_name() );
          scanf("%d",&(in_val[i]));
       }

       printf("Enter values for the %d sensor  signals:\n",out_last);
       for(i=0;i<out_last;++i) {
          printf("Sensor    %s:",get_signal_rec(get_sensor(i)).get_name() );
          scanf("%d",&(rec_val[i]));
       }
    }
    else {
       if(is_flag_set(PRINT_TRIV)) {
          puts("Getting commands,sensors");
       }
       for(i=0;i<in_last;++i) {
          fscanf(confile,"%d",&(in_val[i]));
          if(is_flag_set(PRINT_TRIV)) {
             printf("Command   %s: %d\n",get_signal_rec(get_command(i)).get_name(),
                    in_val[i] );
          }
       }
       for(i=0;i<out_last;++i) {
          fscanf(confile,"%d",&(rec_val[i]));
          if(is_flag_set(PRINT_TRIV)) {
             printf("Sensor    %s: %d\n",get_signal_rec(get_sensor(i)).get_name(),
                    rec_val[i] );
          }
       }
    }

    diagnose( in_val,in_last,out_val,out_last,rec_val);
}

//=========================================================================
```

## F.8 CODE.HPP

```
//
//
// CODE.HPP
//
// Code block class
//
// 16 July 1992
//-----------------------------------------------------------------
#ifndef __CODE_HPP__
#define __CODE_HPP__

#include "mcode.hpp"

#define MAX_CODE_LEN    40 // Max length of op codes
#define MAX_CODE_BLOCKS 40 // Max number of code blocks

class Code {
public:
   Code(void);
   Code(int new_id);
   int   get_id(void);              // Return ID for code
   void  add_mcode(MCode new_code); // Add mcode to code object
   void  execute(int bi_no);        // Execute Code block
   void  print(char *s);            // Print code description

   int   get_code_blk_len(void);    // Return length of code
   MCode get_mcode_at(int pos);     // Get mcode from code block
private:
   int   id;                      // Code id number
   MCode code_blk[MAX_CODE_LEN]; // MCode storage
   int   last_code_no;            // Last MCode
};
//-----------------------------------------------------------------
// Code block storage management routines

void  reset_code_block(void);
void  add_code_block( Code &new_code);
Code &get_code_block(int id);
void  code_block_print(char *s);
//-----------------------------------------------------------------
// Execution stack routines

void value_reset(void);
void value_push(int value);
int  value_pop(void);

//-----------------------------------------------------------------
#endif
```

## F.9 CODE.CPP

```
//
//
// CODE.CPP
//
// Code block class
//
// 16 July 1992
//----------------------
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <assert.h>

#include "code.hpp"
#include <stacks.h>

typedef BI_StackAsVector<int> intStack;

void  Code::execute(int bi_no)
{
   int      code_pc = 0;   // mcode PC
   int      not_done = TRUE;

   //  printf("Code::Execute: Behave #%2d\n",bi_no);
   value_reset();
   while(not_done) {
      not_done = ((code_blk[code_pc]).execute(bi_no));
      code_pc++;
   }
}
//-----------------------------------------------------------------------

typedef BI_StackAsVector<int> intStack;

intStack value_stack;

/* Clear any remaining values on value stack */
void value_reset(void)
{
   while( !value_stack.isEmpty() ) {
      value_stack.pop();
   }
}
void value_push(int value)
{
   value_stack.push(value);
}
int  value_pop(void)
{
   return value_stack.pop();
```

```
}

//---------------------------------------------------------------------
//---------------------------------------------------------------------
// Storage for Code blocks

static Code code_storage[MAX_CODE_BLOCKS];
static int  last_code_block;


void reset_code_block(void)
{
   last_code_block = 0;
}

void add_code_block( Code &new_code)
{
   assert(last_code_block<MAX_CODE_BLOCKS);
   code_storage[last_code_block++] = new_code;
}

Code &get_code_block(int id)
{
   for(int i=0; i<last_code_block; ++i) {
      if( code_storage[i].get_id() == id ) {
         return code_storage[i];
      }
   }
   puts("!!!!!! Bad id in get_code_block---code.cpp");
   exit(119);
   return code_storage[0];
}

void code_block_print(char *s)
{
   printf(s);
   for(int i=0; i<last_code_block; ++i) {
      printf("-----Code block %2d-----\n",i);
      code_storage[i].print("");
      getch();
   }
   puts("-----------------------");
}


//=====================================================================
Code::Code(void)
{
   id = -1;
   last_code_no = 0;
}
```

```
//---------------------------------------------------------------------
Code::Code(int new_id)
{
   id = new_id;
   last_code_no = 0;
}
//---------------------------------------------------------------------
int   Code::get_id(void)
{
   return id;
}
//---------------------------------------------------------------------
void  Code::add_mcode(MCode new_code)
{
   assert(last_code_no<MAX_CODE_LEN);
   code_blk[last_code_no++] = new_code;
}
//---------------------------------------------------------------------
void Code::print(char *s)
{
   printf(s);
   printf("For Code_block %2d, MCodes are:\n",id);
   for(int i=0;i<last_code_no;++i) {
      printf("%2d| ",i);
      code_blk[i].print();
      puts("");
   }
}
//---------------------------------------------------------------------
int   Code::get_code_blk_len(void)
{
   return last_code_no;
}
//---------------------------------------------------------------------
MCode Code::get_mcode_at(int pos)
{
   return code_blk[pos];
}
//---------------------------------------------------------------------
//=====================================================================
```

*F.10 COMSEN.HPP*

```
//
//
// COMSEN.HPP
//
// Handle Command and sensor lists
// 16 jul 92
//
//--------------------------
//-------------------------------------------------------------------

#define MAX_COMMANDS 20 // Max number of commands (system inputs)
#define MAX_SENSORS  20 // Max number of sensors  (system outputs)

#define COMMAND_SR  -1  // Signal is driven by a Command
#define SENSOR_SR   -2  // Signal drives a sensor
//-------------------------------------------------------------------
void reset_commands(void);         // Initialize
void add_command(int signal_id);   // Add new command to list
int  get_command(int command_no);  // Get command
int  get_last_command(void);       // Get the last command in the list

void reset_sensors(void);          // Initialize
void add_sensor(int signal_id);    // Add new sensor to list
int  get_sensor(int sensor_no);    // Get sensor
int  get_last_sensor(void);        // Get the last sensor in the list

//-------------------------------------------------------------------
```

## F.11 COMSEN.CPP

```
//
//
// COMSEN.CPP
//
// Handle Command and sensor lists
// 16 jul 92
//
//--------------------------

#include <assert.h>

#include "comsen.hpp"

//-----------------------------------------------------------------------
// Storage for Commands

static int  commands[MAX_COMMANDS];
static int  last_command;

// Storage for sensors

static int  sensors[MAX_SENSORS];
static int  last_sensor;
//-----------------------------------------------------------------------

void reset_commands(void)
{
   last_command = 0;
}
void add_command(int signal_id)
{
   assert(last_command < MAX_COMMANDS);
   commands[last_command++] = signal_id;
}
int  get_command(int command_no)
{
   assert((command_no>=0) && (command_no<last_command));
   return commands[command_no];
}
int  get_last_command(void)
{
   return last_command;
}

//-----------------------------------------------------------------------

void reset_sensors(void)
{
   last_sensor = 0;
}
```

```
void add_sensor(int signal_id)
{
   assert(last_sensor < MAX_SENSORS);
   sensors[last_sensor++] = signal_id;
}
int  get_sensor(int sensor_no)
{
   assert((sensor_no>=0) && (sensor_no<last_sensor));
   return sensors[sensor_no];
}
int  get_last_sensor(void)
{
   return last_sensor;
}

//-----------------------------------------------------------------
```

## F.12 INT.HPP

```
//
// INT.HPP
//
// Integer Class - for IDs
//
// This object puts a shell around an integer. It is required
// so that integers can be used with the Borland container library
//
//
#ifndef __INT_HPP__
#define __INT_HPP__ 1

#include <object.h>
//
#define IntegerClass 111
class Integer : public Object {
public:
   Integer(int new_data = 0) {
      data = new_data;
   }
   int value(void) { return data; }

   virtual hashValueType hashValue() const {
      return data;
   }

   virtual int isEqual( const Object& otherObj ) const {
      return data == ((Integer&) otherObj).data;
   }

   virtual int isLessThan( const Object& otherObj ) const {
      return data < ((Integer &) otherObj).data;
   }

   virtual classType isA() const {return IntegerClass;}

   virtual char *nameOf() const {return "Integer";}

   virtual void printOn( ostream& coutt ) const {
      coutt << "Int: " << data;
   }
private:
   int   data;
};
//
#endif
```

*F.13 MAIN.CPP*

```
//
//
// MAIN.CPP - main function and supporting routines
//
// 24 Aug 92
//
// This is the " in module of Calvin. This module handles initializing'
// Calvin. The VHDL code is the parsed.  Control is then handed
// to the diagnostic moduiem CALVIN.CPP.
//---------------------------
//
#define __MAIN_CPP__

#include <stdio.h>
#include <conio.h>
#include <assert.h>
#include <string.h>
#include <stdlib.h>

#include "thesis.h"
#include "vhdl.hpp"
#include "stat.hpp"

#define ERR_STR_LEN 128  // Length of error string argument parsing

//FILE *infile;   // input file for source code
//FILE *confile;  // input file for commands
//int   G_code;   // Code flag for output options

//-----------------------------------------------------------------
// PRINT_TRIV -
//    Print out headings,etc.
// PRINT_HYPO - T.CPP
//    "selecting fault condition %d"
//    "Ruled out hypothesis %d"
//    "For %d, we have %d bodies"
// PRINT_COMM - UV.LEX
//    Print out comments
// PRINT_1SIM - T.CPP
//    print out results from first simulation
// PRINT_VH" - VHDL.CPP
//    many
// PRINT_SUSP - T.CPP
//    print list of suspects after collect_bi_suspects()
// INSERT_BRK - T.cpp
//    break inserted as soon as error found and diagnosed
// COLLECT__2 - T.cpp
//    replace collect_bi_suspects() with collect_bi_suspects2()
```

```
int flags[MAX_FLAG]; // System flags set by command line

int is_flag_set(int flag_no)
{
   return( flags[flag_no] );
}

void set_flag(char *flag_str)
{
   for(int i=0;i<MAX_FLAG;++i) {
      switch( *(flag_str+i) ) {
      case '\0':
         yyerror("Missing Flag on command line");
         exit(99);
      case '1':
         flags[i] = 1;
         break;
      case '0':
         flags[i] = 0;
         break;
      default:
         yyerror("Illegal flag on command line");
         exit(99);
      }
   }
}
//-----------------------------------------------------------------
void print_system_flags(void)
{
   puts("System flag status:");
   printf("PRINT_TRIV:  %d\n", flags[PRINT_TRIV] );
   printf("PRINT_HYPO:  %d\n", flags[PRINT_HYPO] );
   printf("PRINT_COMM:  %d\n", flags[PRINT_COMM] );
   printf("PRINT_1SIM:  %d\n", flags[PRINT_1SIM] );
   printf("PRINT_VHDL:  %d\n", flags[PRINT_VHDL] );
   printf("PRINT_SUSP:  %d\n", flags[PRINT_SUSP] );
   printf("INSERT_BRK:  %d\n", flags[INSERT_BRK] );
   printf("COLLECT__2:  %d\n", flags[COLLECT__2] );
   puts("--------------------");
}
//------------ ---------------------------------------------------
void parse_code_flag( int argc, char **argv)
{
   char *file_name;
   char err_str[ERR_STR_LEN];

   if(argc < 2 ) {
      puts("Usage:");
      puts("   CALVIN ffffffff vhdl_file input_file\n");
      puts("where:
      puts("     ffffffff   - Calvin flags (0/1)");
```

```
      puts("     vhdl_file  - VHDL source code file");
      puts("     input_file - Optional input file for in/out values\n");
      puts("Flags:");
      puts("    0 - Print trivia");
      puts("    1 - Print hypotheses");
      puts("    2 - Print commands");
      puts("    3 - Print results of first simulation");
      puts("    4 - Print VHDL");
      puts("    5 - Print suspects");
      puts("    6 - Insert breakpoint");
      puts("    7 - Use Collection");
      exit(111);
   }
   else {
      set_flag(*(argv+1));
   }
   if(is_flag_set(PRINT_TRIV)) {
      print_system_flags();
   }
}
//-------------------------------------------------------------------

void parse_source_file( int argc, char **argv)
{
   char *file_name;
   char err_str[ERR_STR_LEN];

   if(argc < 3 ) {
      puts("Source file not found; using \"ttt\"");
      file_name = "ttt";
   }
   else {
      file_name = *(argv+2);
   }
   if( (infile=fopen(file_name,"r")) == NULL ) {
      sprintf(err_str,"Cannot open source file \"%s\"", file_name);
      yyerror(err_str);
      exit(112);
   }
}
//-------------------------------------------------------------------

void parse_input_file( int argc, char **argv)
{
   char *file_name;
   char err_str[ERR_STR_LEN];

   if(argc < 4 ) {
      puts("input file not found; using \"con\"");
      file_name  = "con";
      G_con_flag = TRUE;
```

```
      return;
   }
   else {
      file_name  = *(argv+3);
      G_con_flag = FALSE;
   }
   if( (confile=fopen(file_name,"r")) == NULL ) {
      sprintf(err_str,"Cannot open source file \"%s\"", file_name);
      yyerror(err_str);
      exit(113);
   }
   // Determine number of inputs lines to process
   if(fscanf(confile,"%d",&G_no_inputs) != 1 ) {
      yyerror("Syntax error while reading confile header");
      exit(114);
   }
   // Eat CRLF at end of line
   fgets(err_str, ERR_STR_LEN, confile);
}
//---------------------------------------------------------------------
void main(int argc, char **argv)
{
   puts("VHDL Diagnostic System");
   init_sim();
   parse_code_flag(argc,argv);
   parse_source_file(argc,argv);
   parse_input_file(argc,argv);
   yyparse();
   if(G_con_flag) {
      reset_stats();
      run_exam();
      print_stats();
      if(is_flag_set(PRINT_TRIV)) {
         puts("=============END=OF=RUN====================");
      }
   }
   else {
      for(int i=0; i<G_no_inputs; ++i) {
         reset_stats();
         run_exam();
         print_stats();
         if(is_flag_set(PRINT_TRIV)) {
            puts("=============END=OF=RUN============================");
         }
      }
   }
   puts("El Fin.");
}
//---------------------------------------------------------------------
```

## F.14 MCODE.HPP

```
//
//
// MCODE.HPP
//
// Microcode class
//
// Handle execution of individual mcodes.
// 16 July 1992
//----------------------
#ifndef __MCODE_HPP__
#define __MCODE_HPP__

//-----------------------------------------------------------------
#define MAX_STORE_LEN 20  // Number of temp store locations

// MCodes implemented

#define M_NULL  -1    // Null opcode
#define M_GET   -2    // Get signal  (signal #)
#define M_POST  -3    // Post signal (signal #, value, delta time)
#define M_PUSH  -4    // Push??
#define M_NOT   -5    // NOT  (value)
#define M_AND   -6    // AND  (value1, value2)
#define M_OR    -7    // OR   (value1, value2)
#define M_XOR   -8    // XOR  (value1, value2)
#define M_END   -9    // End execution
#define M_NAND  -10   // NAND (value1, value2)
#define M_NOR   -11   // NAND (value1, value2)

#define M_POP   -12   // Pop (and discard) value on top of stack
#define M_STORE -13   // Store (addr) -- Place TOS in temp store
#define M_RETRV -14   // Retrieve (addr) -- Place value from store on TOS

class MCode {
public:
   MCode(void);               // Create null microcode
   MCode(int new_op);         // Create new microcode
   int   execute(int bi_no);  // Execute the opcode
   void  print(void);         // print translation of opcode
   int   get_op_code(void);   // Return mcode op code
private:
   int   op_code;
};
//-----------------------------------------------------------------
#endif
```

## F.15 MCODE.CPP

```
//
//
// MCODE.CPP
//
// Microcode class
//
// 16 July 1992
//
// This module handle the microcode execution.
//---------------------
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <assert.h>

#include "mcode.hpp"
#include "code.hpp"
#include "behave.hpp"
#include "signal.hpp"
#include "vhdl.hpp"
//-----------------------------------------------------------------------
int exec_null(void);
int exec_get(int bi_no);
int exec_post(int bi_no);
int exec_push(void);
int exec_not(void);
int exec_and2(void);
int exec_or2(void);
int exec_xor2(void);
int exec_end(void);
int exec_pop(void);
int exec_store(void);
int exec_retrieve(void);
//-----------------------------------------------------------------------
// Temp store for values during code execution
int G_store[MAX_STORE_LEN];

//-----------------------------------------------------------------------
MCode::MCode(void)
{
   op_code = M_NULL;
}
//-----------------------------------------------------------------------
MCode::MCode(int new_op)
{
   op_code = new_op;
}
//-----------------------------------------------------------------------
MCode::get_op_code(void)
{
```

```
   return op_code;
}
//-----------------------------------------------------------------------
int  MCode::execute(int bi_no)
{
   // printf("Executing for bi#%2d: ",bi_no);
   // print();
   // puts("");
   switch(op_code) {
   case M_NULL:
      return exec_null();
   case M_GET:
      return exec_get(bi_no);
   case M_POST:
      return exec_post(bi_no);
   case M_PUSH:
      return exec_push();
   case M_NOT:
      return exec_not();
   case M_AND:
      return exec_and2();
   case M_OR:
      return exec_or2();
   case M_XOR:
      return exec_xor2();
   case M_END:
      return exec_end();
   case M_POP:
      return exec_pop();
   case M_STORE:
      return exec_store();
   case M_RETRV:
      return exec_retrieve();
   default:
      if(op_code >= 0) {
         value_push(op_code);
         return TRUE;
      }
      else {
         printf("***** Illegal opcode %2d !!!!!\n",op_code);
         exit(122);
      }
   }
   return FALSE;
}
//-----------------------------------------------------------------------
void  MCode::print(void)
{
   switch(op_code) {
   case M_NULL:
      printf("M_NULL");
```

```
      break;
   case M_GET:
      printf("M_GET");
      break;
   case M_POST:
      printf("M_POST");
      break;
   case M_PUSH:
      printf("M_PUSH");
      break;
   case M_NOT:
      printf("M_NOT");
      break;
   case M_AND:
      printf("M_AND");
      break;
   case M_OR:
      printf("M_OR");
      break;
   case M_XOR:
      printf("M_XOR");
      break;
   case M_END:
      printf("M_END");
      break;
   case M_POP:
      printf("M_POP");
      break;
   case M_STORE:
      printf("M_STORE");
      break;
   case M_RETRV:
      printf("M_RETRV");
      break;
   default:
      if(op_code >= 0) {
         printf("Value: %2d",op_code);
      }
      else {
         printf("*Unknown*");
      }
   }
}
//-----------------------------------------------------------------
//-----------------------------------------------------------------
int exec_null(void)
{
   return TRUE;
}
int exec_get(int bi_no)
{
```

```
    Behave &bi_ptr       = get_behave_inst(bi_no);
    int  signal_offset   = value_pop();
    int  signal_id       = bi_ptr.get_input(signal_offset);
    SignalRecord &sr_ptr = get_signal_rec(signal_id);
    int  value           = sr_ptr.get_cval();
    value_push(value);
    return TRUE;
}
int exec_post(int bi_no)
{
    int  time_offset     = value_pop();
    int  value           = value_pop();
    int  signal_offset   = value_pop();
    Behave &bi_ptr       = get_behave_inst(bi_no);
    int  signal_id       = bi_ptr.get_output(signal_offset);
    SignalRecord &sr_ptr = get_signal_rec(signal_id);
    post_signal(get_current_time()+time_offset,sr_ptr.get_id(),value);
    return TRUE;
}
int exec_push(void)
{
    puts("!!!!! UNIMPLEMENTED OP CODE IN MCODE");
    return FALSE;
}
int exec_not(void)
{
    int value = value_pop();
    value_push(!value);
    return TRUE;
}
int exec_and2(void)
{
    int value1 = value_pop();
    int value2 = value_pop();
    value_push(value1 & value2);
    return TRUE;
}
int exec_or2(void)
{
    int value1 = value_pop();
    int value2 = value_pop();
    value_push(value1 | value2);
    return TRUE;
}
int exec_xor2(void)
{
    int value1 = value_pop();
    int value2 = value_pop();
    int value3 = ((value1 & !value2) | (!value1 & value2));
    value_push(value3);
    return TRUE;
```

```
}
int exec_end(void)
{
   return FALSE;
}

int exec_pop(void)
{
   value_pop();
   return TRUE;
}

int exec_store(void)
{
   int addr = value_pop();
   assert(addr >= 0 && addr < MAX_STORE_LEN);
   G_store[addr] = value_pop();
   return TRUE;
}

int exec_retrieve(void)
{
   int addr = value_pop();
   assert(addr >= 0 && addr < MAX_STORE_LEN);
   value_push(G_store[addr]);
   return TRUE;
}

//-----------------------------------------------------------------
//-----------------------------------------------------------------
```

*F.16 SIGNAL.HPP*

```
//
//
// SIGNAL.HPP
//
// Signal Record Class
//
// 16 July 1992
//---------------------

#define MAX_NAME_SIZE  10 // Max size of name
#define MAX_CONNS      10 // Max number of behaves the signal can drive

#define MAX_SIGNAL_REC 50 // Max number of signals in simulation

class SignalRecord {
public:
   SignalRecord(void);
   SignalRecord(
      int   new_id,
      char *new_name,
      int   driver_bi_no);

   get_id(void);              // Get id of signal
   void print(char *s);       // Print signal description
   void add_conns(            // Add connection to signal
      int conns_id);
   void get_conns(            // Get a Behave object connected to signal
      int *last_conn_no,
      int **conn_list);
   void set_cval(int new_val);// Set signal value
   int  get_cval(void);       // Get signal's value
   int  get_driver_bi(void);  // Get ID of Behave object that drives this
                              // signal
   void set_driver_bi(        // Set driver Behave object for this signal.
      int new_driver);
   char *get_name(void);      // Return character name of this signal
private:
   int   id;                  // Integer name
   char  name[MAX_NAME_SIZE]; // Character name
   int   cval;                // Value of signal
   int   conns[MAX_CONNS];    // List of connections to signal (BI inputs)
   int   last_conn;           // Last added conn + 1
   int   driver_bi;           // Which bi# drivers this signal
};

//=============================================================
// Signal record storage management routines

void           reset_signal_rec(void);
void           add_signal_rec( SignalRecord &new_signal);
```

```
void            mod_signal_rec( SignalRecord &mod_signal);
SignalRecord   &get_signal_rec(int id);
void            signal_rec_print(char *s);
//=========================================================================
```

## F.17 SIGNAL.CPP

```
//
//
// SIGNAL.HPP
//
// Signal Record Class
//
// 16 July 1992
//
// This module defines routines for the signal class
//----------------------

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

#include "signal.hpp"
//-------------------------------------------------------------------
// Storage for signal instance

static SignalRecord signal_storage[MAX_SIGNAL_REC];
static int  last_signal_inst;


//-------------------------------------------------------------------
void reset_signal_rec(void)
{
   last_signal_inst = 0;
}
//-------------------------------------------------------------------
void add_signal_rec( SignalRecord &new_signal)
{
   assert(last_signal_inst<MAX_SIGNAL_REC);
   signal_storage[last_signal_inst++] = new_signal;
}
//-------------------------------------------------------------------
void mod_signal_rec( SignalRecord &mod_signal)
{
   for(int i=0; i<last_signal_inst; ++i) {
      if( signal_storage[i].get_id() == mod_signal.get_id() ) {
         signal_storage[i] = mod_signal;
         return;
      }
   }
   puts("!!!!!! Bad id in mod_signal_rec---signal.cpp");
   exit(120);
}
//-------------------------------------------------------------------
SignalRecord &get_signal_rec(int id)
{
```

```
   for(int i=0; i<last_signal_inst; ++i) {
      if( signal_storage[i].get_id() == id ) {
         return signal_storage[i];
      }
   }
   puts("!!!!!! Bad id in get_signal_rec---signal.cpp");
   exit(121);
   return signal_storage[0]; // Get rid of "Need return" warning
}
//---------------------------------------------------------------------
void signal_rec_print(char *s)
{
   printf(s);
   for(int i=0; i<last_signal_inst; ++i) {
      printf("Signal %2d: %03d ==> %d\n",i,
      signal_storage[i].get_id(),
      signal_storage[i].get_cval() );
   }
   puts("-----------------------");
}
//=====================================================================
SignalRecord::SignalRecord(void)
{
   id        = -1;
   strcpy(name,"ANON");
   driver_bi = -1;
   last_conn = 0;
   cval      = 0;
}
//---------------------------------------------------------------------
SignalRecord::SignalRecord(
   int   new_id,
   char *new_name,
   int   driver_bi_no)
{
   id = new_id;
   strncpy(name,new_name,MAX_NAME_SIZE);
   name[MAX_NAME_SIZE] = '\0';
   driver_bi = driver_bi_no;
   last_conn = 0;
   cval      = 0;
}
//---------------------------------------------------------------------
int SignalRecord::get_id(void)
{
   return id;
}
//---------------------------------------------------------------------
void SignalRecord::print(char *s)
{
   printf(s);
```

```
   printf("Name: %10s  (%2d)  Driver: %2d  Value: %2d\n",
      name,id,driver_bi,cval);
   puts("Connected to:");
   for(int i=0;i<last_conn;++i) {
      printf("%2d| --> %2d\n",i,conns[i]);
   }
}
//---------------------------------------------------------------------
char * SignalRecord::get_name(void)
{
   return name;
}
//---------------------------------------------------------------------
void SignalRecord::add_conns(int conns_id)
{
   assert( last_conn<MAX_CONNS  ;

   conns[last_conn++] = conns_id;
}
//---------------------------------------------------------------------
void SignalRecord::get_conns(int *last_conn_no, int **conn_list)
{
   *last_conn_no = last_conn;
   *conn_list    = conns;
}
//---------------------------------------------------------------------
void SignalRecord::set_cval(int new_val)
{
   cval = new_val;
}
//---------------------------------------------------------------------
int  SignalRecord::get_cval(void)
{
   return cval;
}
//---------------------------------------------------------------------
int  SignalRecord::get_driver_bi(void)
{
   return driver_bi;
}
//---------------------------------------------------------------------
void SignalRecord::set_driver_bi(int new_driver)
{
   driver_bi = new_driver;
}
//=====================================================================
```

*F.18 STAT.HPP*

```
//
//
// STAT.HPP - Statistic collection routines
//
// 26 Aug 92
//
//---------------------------
//
// Available statistics
#define NO_SUSPECTS     0
#define NO_HYPO_CHECKED 1
#define NO_FAULTS_FOUND 2
#define NO_POST_SIG     3
#define NO_UPDATE       4
#define NO_VHDL_SIM     5

#define MAX_STAT 6


void reset_stats(void);
void inc_stat(int stat_name);
void print_stats(void);
```

## F.19 STAT.CPP

```
//
//
// STAT.CPP - Statistic collection routines
//
// 26 Aug 92
//
//--------------------------
//
#include <stdio.h>
#include <conio.h>
#include "thesis.h"
#include "stat.hpp"

// collection variables
static int stats[MAX_STAT];


// Reset stat variables
void reset_stats(void)
{
   for( int i=0; i<MAX_STAT; ++i) {
      stats[i] = 0;
   }
}

// increment stat variable
void inc_stat(int stat_name)
{
   stats[stat_name]++;
}

// print stats
void print_stats(void)
{
   if(is_flag_set(PRINT_TRIV)) {
      puts  ("------------ Statistics -------------");
      printf("Number of suspects generated ----- %3d\n", stats[NO_SUSPECTS]);
      printf("Number of hypotheses checked ----- %3d\n", stats[NO_HYPO_CHECKED]);
      printf("Number of faults found ----------- %3d\n", stats[NO_FAULTS_FOUND]);
      printf("Number of posted signals --------- %3d\n", stats[NO_POST_SIG]);
      printf("Number of behave updates --------- %3d\n", stats[NO_UPDATE]);
      printf("Number of simulations done ------- %3d\n", stats[NO_VHDL_SIM]);
      puts  ("-------------------------------------");
   }
   else {
      puts("-----");
      printf("%3d #suspects\n", stats[NO_SUSPECTS]);
      printf("%3d #hypos\n",     stats[NO_HYPO_CHECKED]);
      printf("%3d #faults\n",    stats[NO_FAULTS_FOUND]);
      printf("%3d #posts\n",     stats[NO_POST_SIG]);
```

```
        printf("%3d #updates\n",    stats[NO_UPDATE]);
        printf("%3d #sims\n",       stats[NO_VHDL_SIM]);
    }
}
```

*F.20 THESIS.H*

```
/*
   VHDL PARSER

   File: THESIS.H

   Date: 2 July 1992

   Catch-all file for all modules

*/
/*-----------------------------------------------------------------*/
#ifndef __THESIS_H__
#define __THESIS_H__

#define CURRENT_LIST (void *)1
#define ERROR        -32767
#define TRUE          1
#define FALSE         0

// Code title describing correctly operating code blocks
#define CORRECT_CODE_TITLE "Correct operation"

// Maximum interface parameters for Behave objects
#define MAX_IN   10 // Maximum number of inputs
#define MAX_OUT  10 // Maximum number of outputs
/*-----------------------------------------------------------------*/
int    yyerror(char *s);
int    yylex(void);
int    yyparse(void);

void   run_exam(void);
int    is_flag_set(int flag_no);

//void *alloca();
/*-----------------------------------------------------------------*/
// System flags

#define PRINT_TRIV 0 // Print out headings,etc.
#define PRINT_HYPO 1 // Print out hypothesis numbers
#define PRINT_COMM 2 // Print comment lines during parse
#define PRINT_1SIM 3 // Print out 1st simulation resuts (correct operation)
#define PRINT_VHDL 4 // Print out VHDL output during simulation
#define PRINT_SUSP 5 // Print possible-suspect-list
#define INSERT_BRK 6 // Insert break after one error found
#define COLLECT__2 7 // Use 2nd collect_bi_suspects()

#define MAX_FLAG   8

/*-----------------------------------------------------------------*/
// Global variables
```

```
#include <stdio.h>
#ifdef __MAIN_CPP__
FILE *infile;     // input file for source code
FILE *confile;    // input file for commands
int   G_code;     // Code flag for output options
int   G_con_flag; // Flag to indicate commands come from console
int   G_no_inputs;// Number of command lines to process
#else
extern FILE *infile;     // input file for source code
extern FILE *confile;    // input file for commands
extern int   G_code;     // Code flag for output options
extern int   G_con_flag; // Flag to indicate commands come from console
extern int   G_no_inputs;// Number of command lines to process
#endif
/*-----------------------------------------------------------------*/
#endif
```

*F.21 VHDL.HPP*

```
//
//
// VHDL.HPP
//
// VHDL simulator code
//
// 17 July 1992
//
// Header file for VHDL simulator module
//----------------------
#ifndef __VHDL_HPP__
#define __VHDL_HPP__
//-------------------------------------------------------------------
#define OK          0
#define QUEUE_END   1
//-------------------------------------------------------------------
void init_sim(void);    // Init simulator
void update_behave(     // Simulate a behavior object
   int &behave_id,
   void *args);
void post_signal(       // Post an activation record to the queue.
   int   time,          // This function called during behave object
   int   signal_id,     // simulation
   int   new_val );
int  get_top_time(void);   // Get time of next event in the queue
int get_current_time(void);// Get the current simulation time
int process_low_time(void);// Process all activation records with the
                           // current simulation time
int process_init(void);    // Execute each behavior object once in
                           // order to start the simulation object
vhdl_main_loop(void);      // VHDL main loop - called to run the
                           // VHDL simulation
//-------------------------------------------------------------------
#endif
```

*F.22 VHDL.CPP*

```
//
//
// VHDL.CPP
//
// VHDL simulator code
//
// 17 July 1992
//----------------------
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

#include <sets.h>

#include "signal.hpp"
#include "behave.hpp"
#include "block.hpp"
#include "code.hpp"

#include "ar.hpp"
#include "vhdl.hpp"
#include "stat.hpp"
#include "queue.hpp"

typedef BI_SetAsVector<int> IntSet;
//-----------------------------------------------------------------
static int current_time;
//-----------------------------------------------------------------
void init_sim(void)
{
   current_time = 0;
   reset_behave_inst();
   reset_signal_rec();
   reset_code_block();
   reset_block_inst();
}

static void update_behave(int &behave_id, void *args)
{
   inc_stat(NO_UPDATE);

   if( is_flag_set(PRINT_VHDL)) {
      printf("Updating behave %03d\n",behave_id);
   }

   // Get behavior instance to update
   Behave &bi = get_behave_inst(behave_id);

   // Get proper code block for behavior
   block &block_ptr = get_block_inst(bi.get_block_id());
```

```
   Code  &code_ptr  = get_code_block(block_ptr.get_code(bi.get_current_select()));
   // And execute it
   code_ptr.execute(behave_id);
   // Get rid of Borland warning (do nothing)
   ((int*)(args))++;
}
//-----------------------------------------------------------------
void post_signal(
   int   time,
   int   signal_id,
   int   new_val )
{

   inc_stat(NO_POST_SIG);
   if( is_flag_set(PRINT_VHDL)) {
      printf("Posting signal %03d: new value=%d at %d\n",
         signal_id,new_val,time);
   }
   ActiveRecord new_ar = ActiveRecord(time,signal_id,new_val);
   put_queue(new_ar);

}

//-----------------------------------------------------------------
int get_top_time(void)
{
   if( empty_queue() ) {
      return QUEUE_END;
   }
   ActiveRecord front_ar = front_queue();
   return front_ar.get_time();
}

//-----------------------------------------------------------------
int get_current_time(void)
{
   return current_time;
}
//-----------------------------------------------------------------
int process_low_time(void)
{
   IntSet behav_set;
   int   *conns_ptr;
   int    last_conn_no;
   int    new_time;

   if( (new_time=get_top_time()) == QUEUE_END) {
      return QUEUE_END;
   }
   if( is_flag_set(PRINT_VHDL)) {
      printf("The new time is %d\n", new_time);
```

```
    }

    while(new_time == get_top_time() ) {
       ActiveRecord next_ar = get_queue();
       SignalRecord &sr_ptr = get_signal_rec(next_ar.get_sr_ptr() );

       if(sr_ptr.get_cval() != next_ar.get_value() ) {
          sr_ptr.set_cval( next_ar.get_value() );
          if( is_flag_set(PRINT_VHDL)) {
             printf("Signal %03d: <-- %d\n", sr_ptr.get_id(), sr_ptr.get_cval() );
          }
          // Collect conns into one container
          sr_ptr.get_conns(&last_conn_no, &conns_ptr);
          for(int i=0;i<last_conn_no;++i) {
             if(is_flag_set(PRINT_VHDL)) {
                printf("Will update %03d\n",conns_ptr[i]);
             }
             behav_set.add(conns_ptr[i]);
          }
       }
    }
    // Update master time
    current_time = new_time;

    behav_set.forEach( &update_behave, "");
    return OK;
}
//-----------------------------------------------------------------------
int process_init(void)
{
    int behave_id_no;

    for( int i=0; i<get_last_behave_inst(); ++i ) {
       behave_id_no = get_behave_id_at(i);
       update_behave(behave_id_no,"");
    }
    return OK;
}
//-----------------------------------------------------------------------
vhdl_main_loop(void)
{
    int not_done = TRUE;
    int result;

    inc_stat(NO_VHDL_SIM);
    while(not_done){
       switch(process_low_time()) {
       case OK:
          break;
       case QUEUE_END:
          not_done = FALSE;
```

```
            break;
         default:
            puts("!!!!!Error in VHDL_MAIN_LOOP()");
            exit(110);
         }
      }
      return OK;
}
//-----------------------------------------------------------------------
```

# Appendix G. *Verification of Example VHDL Source Code*

## *G.1 Introduction*

The example source files in Appendix B were verified using the Zycad VHDL system. Because of some limitations in Calvin's VHDL simulator, some modifications were made. These include:

- Libraries were not implemented in Calvin. The "work." and references to the "work" library were added.
- Sensitivity lists were not implemented. These lists were added to the **process** statements.
- Specific time units were not implemented. In Calvin, the times specified in the **after** clauses do not have any units. The unit "ns" was added for the Zycad runs.
- At this time Calvin does not allow signal assignments in the structural descriptions. In some of the circuits internal signals need to be brought out as outputs. Calvin allows the signals in the parameter lists to be used as internal signals. Since this is not allowed by the VHDL standard, new signals were created for the Zycad runs. These can be identified by the letter 'o' at the end of the identifier (as in **i710o**).

In this appendix are the modified source files. These were followed by the signal values as reported by Zycad. For the full-adder, ALU without probes, and ALU with probes, the inputs are the same as the those in the figures in section 4.1.1.

## G.2 Zycad Source Files

### G.2.1 Full-Adder

```
--
-- One-bit full-adder
--
-- Consists of 2 half-adders and an OR gate
--
-- X + Y + Cin = Z + Cout
--
-- This full-adder is used in the four-bit adder
--
--------------------------
-------- OR Gate ---------
entity i015 is
port(
   i011: in  Bit;
   i012: in  bit;
   i013: out bit
);
end;

architecture i025 of i015 is
begin
   process (i011, i012)
   begin
      i013 <= i011 or i012 after 5 ns;
   end process;
end i025;
--------------------------
------ Half adder --------
entity i010 is
port(
   i011: in  Bit;
   i012: in  bit;
   i013: out bit;
   i014: out bit
);
end;

architecture i020 of i010 is
begin
   process (i011,i012)
   begin
      i013 <= i011 xor i012 after 5 ns;
      i014 <= i011 and i012 after 5 ns;
   end process;
end i020;
--------------------------
------ Full Adder --------
```

```
entity i050 is
port(
   i051,i052,i053:in bit;
   i054,i055:out bit
);
end i050;

architecture i060 of i050 is

signal i090:bit;
signal i091:bit;
signal i092:bit;
component i010
   port(
      i011: in  Bit;
      i012: in  bit;
      i013: out bit;
      i014: out bit
   );
end component;
component i030
   port(
      i011,i012:in bit;
      i013:out bit
   );
end component;

begin
   i080:i010
      port map(
         i011 => i051,
         i012 => i052,
         i013 => i090,
         i014 => i091 );

   i081:i010
      port map(
         i011 => i090,
         i012 => i053,
         i013 => i054,
         i014 => i092 );
   i082:i030
      port map(
         i011 => i091,
         i012 => i092,
         i013 => i055 );
end;

-------------------------
------- Circuit ----------
configuration i099 of i050 is
```

```
  for i060
    for i080,i081:i010 use entity work.i010(i020);
    end for;
    for i082:i030 use entity work.i015(i025);
    end for;
  end for;
end;
-------------------------
```

### *G.2.2 ALU without Probes*

```
--
-- Three-bit, Two-operation ALU
-- Performs AND or OR function of 2 three-bit values
--
-- If S=1, A2A1A0 AND B2B1B0 = Z2Z1Z0
-- If S=0, A2A1A0 OR  B2B1B0 = Z2Z1Z0
--
--
-- This example has the probes inserted at the outputs
-- of the AND/OR functions commented out.
--------------------------
library work;
-------- OR Gate ---------
entity i200 is
port(
   i201: in  Bit;
   i202: in  bit;
   i203: out bit
);
end;
architecture i299 of i200 is
begin
   process (i201,i202)
   begin
      i203 <= i201 or i202 after 5 ns;
   end process;
end;
--------------------------
-------- AND Gate ---------
entity i100 is
port(
   i101: in  Bit;
   i102: in  bit;
   i103: out bit
);
end;
architecture i199 of i100 is
begin
   process (i101,i102)
   begin
      i103 <= i101 and i102 after 5 ns;
   end process;
end;
--------------------------
-------- INVGate ---------
entity i300 is
port(
   i301: in  Bit;
   i302: out bit
```

```
);
end;
architecture i399 of i300 is
begin
   process (i301)
   begin
      i302 <= not i301 after 5 ns;
   end process;
end;
--------------------------

entity i500 is
   port(
      i510  : in  bit; -- A
      i511  : in  bit; -- A
      i512  : in  bit; -- A
      i520  : in  bit; -- B
      i521  : in  bit; -- B
      i522  : in  bit; -- B
      i595  : in  bit; -- s0
      i530  : out bit; -- Z
      i531  : out bit; -- Z
      i532  : out bit -- Z
--
-- The following are the commented-out probes
--
--      i710  : out bit; -- Y0AND
--      i711  : out bit; -- Y1AND
--      i712  : out bit; -- Y1AND
--      i810  : out bit; -- Y0OR
--      i811  : out bit; -- Y1OR
--      i812  : out bit  -- Y1OR
     );
end i500;

architecture i599 of i500 is

   component i100
   port( i101,
         i102   :  In    Bit;
         i103   :  out   Bit );
   end component;

   component i200
   port( i201,
         i202   :  In    Bit;
         i203   :  out   Bit );
   end component;

   component i300
   port( i301   :  In    Bit;
```

```
        i302   :  Out   Bit );
   end component;

signal
   i000,
   i001,i003,i004,
   i011,i013,i014,
   i021,i023,i024
   : bit;

--
-- The commented-out probes have been replaced by
-- these internal signals
--
signal i710,i711,i712 : bit;
signal i810,i811,i812 : bit;

begin
   -- Control line inverter
   i606: i300 port map( i301=>i595, i302=>i000 );

   -- Bit 0
   i601: i100 port map( i101=>i510, i102=>i520, i103=>i710 );
   i602: i200 port map( i201=>i510, i202=>i520, i203=>i810 );
   i603: i100 port map( i101=>i710, i102=>i000, i103=>i003 );
   i604: i100 port map( i101=>i810, i102=>i595, i103=>i004 );
   i605: i200 port map( i201=>i003, i202=>i004, i203=>i530 );

   -- Bit 1
   i611: i100 port map( i101=>i511, i102=>i521, i103=>i711 );
   i612: i200 port map( i201=>i511, i202=>i521, i203=>i811 );
   i613: i100 port map( i101=>i711, i102=>i000, i103=>i013 );
   i614: i100 port map( i101=>i811, i102=>i595, i103=>i014 );
   i615: i200 port map( i201=>i013, i202=>i014, i203=>i531 );

   -- Bit 2
   i621: i100 port map( i101=>i512, i102=>i522, i103=>i712 );
   i622: i200 port map( i201=>i512, i202=>i522, i203=>i812 );
   i623: i100 port map( i101=>i712, i102=>i000, i103=>i023 );
   i624: i100 port map( i101=>i812, i102=>i595, i103=>i024 );
   i625: i200 port map( i201=>i023, i202=>i024, i203=>i532 );

end;
--------------------------
------- Circuit ----------
configuration i000 of i500 is
  for i599
    -- AND gates
    for i601,i603,i604:i100 use entity work.i100(i199);
    end for;
```

```
    for i611,i613,i614:i100 use entity work.i100(i199);
    end for;

    for i621,i623,i624:i100 use entity work.i100(i199);
    end for;

    -- OR  gates
    for i602,i605:i200 use entity work.i200(i299);
    end for;

    for i612,i615:i200 use entity work.i200(i299);
    end for;

    for i622,i625:i200 use entity work.i200(i299);
    end for;

    -- INV gates
    for i606:i300 use entity work.i300(i399);
    end for;

  end for;
end;
--------------------------
```

### *G.2.3 ALU with Probes*

```
--
-- Three-bit, Two-operation ALU
-- Performs AND or OR function of 2 three-bit values
--
-- If S=1, A2A1A0 AND B2B1B0 = Z2Z1Z0
-- If S=0, A2A1A0 OR  B2B1B0 = Z2Z1Z0
--
--
-- This example has the probes inserted at the outputs
-- of the AND/OR functions. These bring the results of  both
-- functions to sensors.
--------------------------
library work;

-------- OR Gate ---------
entity i200 is
port(
   i201: in  Bit;
   i202: in  bit;
   i203: out bit
);
end;
architecture i299 of i200 is
begin
   process (i201,i202)
   begin
      i203 <= i201 or i202 after 5 ns;
   end process;
end;
--------------------------
-------- AND Gate ---------
entity i100 is
port(
   i101: in  Bit;
   i102: in  bit;
   i103: out bit
);
end;
architecture i199 of i100 is
begin
   process (i101,i102)
   begin
      i103 <= i101 and i102 after 5 ns;
   end process;
end;
--------------------------
-------- INVGate ---------
entity i300 is
port(
```

```
   i301: in  Bit;
   i302: out bit
);
end;
architecture i399 of i300 is
begin
   process (i301)
   begin
      i302 <= not i301 after 5 ns;
   end process;
end;
--------------------------

entity i500 is
   port(
      i510  : in  bit; -- A
      i511  : in  bit; -- A
      i512  : in  bit; -- A
      i520  : in  bit; -- B
      i521  : in  bit; -- B
      i522  : in  bit; -- B
      i595  : in  bit; -- s0
      i530  : out bit; -- Z
      i531  : out bit; -- Z
      i532  : out bit; -- Z
--
-- These output signals are the probes
--
      i710  : out bit; -- Y0AND
      i711  : out bit; -- Y1AND
      i712  : out bit; -- Y1AND
      i810  : out bit; -- Y0OR
      i811  : out bit; -- Y1OR
      i812  : out bit  -- Y1OR
      );
end i500;

architecture i599 of i500 is

   component i100
   port( i101,
         i102   :  In    Bit;
         i103   :  out   Bit );
   end component;

   component i200
   port( i201,
         i202   :  In    Bit;
         i203   :  out   Bit );
   end component;
```

```
   component i300
   port( i301   :  In    Bit;
         i302   :  Out   Bit );
   end component;

signal
   i000,
   i001,i003,i004,
   i011,i013,i014,
   i021,i023,i024
   : bit;

signal i710o,i711o,i712o,i810o,i811o,i812o : bit;

begin
   -- Control line inverter
   i606: i300 port map( i301=>i595, i302=>i000 );

   -- Bit 0
   i601: i100 port map( i101=>i510, i102=>i520, i103=>i710o );
   i602: i200 port map( i201=>i510, i202=>i520, i203=>i810o );
   i603: i100 port map( i101=>i710o, i102=>i000, i103=>i003 );
   i604: i100 port map( i101=>i810o, i102=>i595, i103=>i004 );
   i605: i200 port map( i201=>i003, i202=>i004, i203=>i530 );

   -- Bit 1
   i611: i100 port map( i101=>i511, i102=>i521, i103=>i711o );
   i612: i200 port map( i201=>i511, i202=>i521, i203=>i811o );
   i613: i100 port map( i101=>i711o, i102=>i000, i103=>i013 );
   i614: i100 port map( i101=>i811o, i102=>i595, i103=>i014 );
   i615: i200 port map( i201=>i013, i202=>i014, i203=>i531 );

   -- Bit 2
   i621: i100 port map( i101=>i512, i102=>i522, i103=>i712o );
   i622: i200 port map( i201=>i512, i202=>i522, i203=>i812o );
   i623: i100 port map( i101=>i712o, i102=>i000, i103=>i023 );
   i624: i100 port map( i101=>i812o, i102=>i595, i103=>i024 );
   i625: i200 port map( i201=>i023, i202=>i024, i203=>i532 );
i710 <= i710o;
i711 <= i711o;
i712 <= i712o;
i810 <= i810o;
i811 <= i811o;
i812 <= i812o;

end;
--------------------------
------- Circuit ----------
configuration i000 of i500 is
  for i599
    -- AND gates
```

```
    for i601,i603,i604:i100 use entity work.i100(i199);
    end for;

    for i611,i613,i614:i100 use entity work.i100(i199);
    end for;

    for i621,i623,i624:i100 use entity work.i100(i199);
    end for;

    -- OR  gates
    for i602,i605:i200 use entity work.i200(i299);
    end for;

    for i612,i615:i200 use entity work.i200(i299);
    end for;

    for i622,i625:i200 use entity work.i200(i299);
    end for;

    -- INV gates
    for i606:i300 use entity work.i300(i399);
    end for;

  end for;
end;
--------------------------
```

### *G.2.4 Four-Bit Adder*

```
--
-- Four-bit Adder
--
-- Consists of 4 full-adders in cascade
--
-- X3X2X1X0 + Y3Y2Y1Y0 + Cin = Z3Z2Z1Z0 + Cout
--
--------------------------
-------- OR Gate ---------
library work;

entity i015 is
port(
   i011: in  Bit;
   i012: in  bit;
   i013: out bit
);
end;

architecture i025 of i015 is
begin
   process (i011,i012)
   begin
      i013 <= i011 or i012 after 5 ns;
   end process;
end i025;
--------------------------
------ Half adder --------
entity i010 is
port(
   i011: in  Bit;
   i012: in  bit;
   i013: out bit;
   i014: out bit
);
end;

architecture i020 of i010 is
begin
   process (i011, i012)
   begin
      i013 <= i011 xor i012 after 5 ns;
      i014 <= i011 and i012 after 5 ns;
   end process;
end i020;
--------------------------
------ Full Adder --------
entity i050 is
port(
```

```
   i100  : in  bit;  -- Cin
   i110,             -- X0
   i111,             -- X1
   i112,             -- X2
   i113  : in  bit;  -- X3
   i120,             -- Y0
   i121,             -- Y1
   i122,             -- Y2
   i123  : in  bit;  -- Y3
   i130,             -- Z0
   i131,             -- Z1
   i132,             -- Z2
   i133  : out bit;  -- Z3
   i140,             -- cout0
   i141,             -- cout1
   i142  : out bit;  -- cout2
   i143  : out bit   -- Cout
);
end;

architecture i060 of i050 is


signal i200,i201,i202:bit;
signal i210,i211,i212:bit;
signal i220,i221,i222:bit;
signal i230,i231,i232:bit;

component i010
   port(
      i011: in  Bit;
      i012: in  bit;
      i013: out bit;
      i014: out bit
   );
end component;
component i030
   port(
      i011,i012:in bit;
      i013:out bit
   );
end component;

signal i140o, i141o, i142o: bit;

begin
-- Bit 0
   i500:i010
      port map(
         i011 => i110,
         i012 => i120,
```

```
            i013 => i200,
            i014 => i201 );

      i501:i010
         port map(
            i011 => i200,
            i012 => i100,
            i013 => i130,
            i014 => i202 );

      i502:i030
         port map(
            i011 => i202,
            i012 => i201,
            i013 => i140o );

   -- Bit 1
      i510:i010
         port map(
            i011 => i111,
            i012 => i121,
            i013 => i210,
            i014 => i211 );

      i511:i010
         port map(
            i011 => i210,
            i012 => i140o,
            i013 => i131,
            i014 => i212 );

      i512:i030
         port map(
            i011 => i212,
            i012 => i211,
            i013 => i141o );
   -- Bit 2
      i520:i010
         port map(
            i011 => i112,
            i012 => i122,
            i013 => i220,
            i014 => i221 );

      i521:i010
         port map(
            i011 => i220,
            i012 => i141o,
            i013 => i132,
            i014 => i222 );
```

```
   i522:i030
      port map(
         i011 => i222,
         i012 => i221,
         i013 => i142o );
-- Bit 3
   i530:i010
      port map(
         i011 => i113,
         i012 => i123,
         i013 => i230,
         i014 => i231 );

   i531:i010
      port map(
         i011 => i230,
         i012 => i142o,
         i013 => i133,
         i014 => i232 );

   i532:i030
      port map(
         i011 => i232,
         i012 => i231,
         i013 => i143 );
i140 <= i140o;
i141 <= i141o;
i142 <= i142o;

end;

---------------------------
------- Circuit ----------
configuration i099 of i050 is
  for i060
    for i500,i501:i010 use entity work.i010(i020);
    end for;
    for i502:i030 use entity work.i015(i025);
    end for;

    for i510,i511:i010 use entity work.i010(i020);
    end for;
    for i512:i030 use entity work.i015(i025);
    end for;

    for i520,i521:i010 use entity work.i010(i020);
    end for;
    for i522:i030 use entity work.i015(i025);
    end for;

    for i530,i531:i010 use entity work.i010(i020);
```

```
    end for;
    for i532:i030 use entity work.i015(i025);
    end for;
  end for;
end;
--------------------------
```

## G.3 Zycad Results

### G.3.1 FULLADD.VHZ

```
i051            '0'
i052            '0'
i053            '0'
i054            '0'
i055            '0'

i051            '1'
i052            '0'
i053            '0'
i054            '1'
i055            '0'
#
i051            '0'
i052            '1'
i053            '0'
i054            '1'
i055            '0'
#
i051            '1'
i052            '1'
i053            '0'
i054            '0'
i055            '1'
#
i051            '0'
i052            '0'
i053            '1'
i054            '1'
i055            '0'
#
i051            '1'
i052            '0'
i053            '1'
i054            '0'
i055            '1'
#
i051            '0'
i052            '1'
i053            '1'
i054            '0'
i055            '1'
#
i051            '1'
i052            '1'
i053            '1'
i054            '1'
```

```
i055            '1'
#
```

### *G.3.2 ALU.VHZ (without Probes)*

```
i510            '1'
i520            '1'
i511            '1'
i521            '1'
i512            '1'
i522            '1'
i595            '0'
i530            '1'
i531            '1'
i532            '1'
#

i510            '1'
i520            '0'
i511            '1'
i521            '1'
i512            '1'
i522            '1'
i595            '0'
i530            '0'
i531            '1'
i532            '1'
#
i510            '1'
i520            '0'
i511            '1'
i521            '1'
i512            '1'
i522            '1'
i595            '1'
i530            '1'
i531            '1'
i532            '1'
#
i510            '1'
i520            '0'
i511            '0'
i521            '1'
i512            '0'
i522            '1'
i595            '1'
i530            '1'
i531            '1'
i532            '1'
#
i510            '1'
i520            '0'
i511            '0'
i521            '1'
```

```
i512            '0'
i522            '1'
i595            '0'
i530            '0'
i531            '0'
i532            '0'
#
```

### G.3.3 ALU1.VHZ (with Probes)

```
i595            '0'
i510            '1'
i511            '1'
i512            '1'
i520            '1'
i521            '1'
i522            '1'
i530            '1'
i531            '1'
i532            '1'
i710            '1'
i810            '1'
i711            '1'
i811            '1'
i712            '1'
i812            '1'
#
i595            '0'
i510            '1'
i511            '1'
i512            '1'
i520            '0'
i521            '1'
i522            '1'
i530            '0'
i531            '1'
i532            '1'
i710            '0'
i810            '1'
i711            '1'
i811            '1'
i712            '1'
i812            '1'
#
i595            '1'
i510            '1'
i511            '1'
i512            '1'
i520            '0'
i521            '1'
i522            '1'
i530            '1'
i531            '1'
i532            '1'
i710            '0'
i810            '1'
i711            '1'
i811            '1'
i712            '1'
```

```
i812            '1'
#
i595            '1'
i510            '1'
i511            '0'
i512            '0'
i520            '0'
i521            '1'
i522            '1'
i530            '1'
i531            '1'
i532            '1'
i710            '0'
i810            '1'
i711            '0'
i811            '1'
i712            '0'
i812            '1'
#
i595            '0'
i510            '1'
i511            '0'
i512            '0'
i520            '0'
i521            '1'
i522            '1'
i530            '0'
i531            '0'
i532            '0'
i710            '0'
i810            '1'
i711            '0'
i811            '1'
i712            '0'
i812            '1'
#
```

```
i100            '0'
i110            '0'
i111            '0'
i112            '0'
i113            '0'
i120            '0'
i121            '0'
i122            '0'
i123            '0'
i130            '0'
i131            '0'
i132            '0'
i133            '0'
i140            '0'
i141            '0'
i142            '0'
i143            '0'
#
i100            '1'
i110            '1'
i111            '0'
i112            '0'
i113            '0'
i120            '1'
i121            '0'
i122            '0'
i123            '0'
i130            '1'
i131            '1'
i132            '0'
i133            '0'
i140            '1'
i141            '0'
i142            '0'
i143            '0'
#
i100            '1'
i110            '1'
i111            '1'
i112            '1'
i113            '1'
i120            '1'
i121            '0'
i122            '0'
i123            '0'
i130            '1'
i131            '0'
i132            '0'
```

```
i133            '0'
I140            '1'
I141            '1'
I142            '1'
i143            '1'
#
i100            '1'
i110            '1'
i111            '1'
i112            '1'
i113            '1'
i120            '1'
i121            '1'
i122            '1'
i123            '1'
i130            '1'
i131            '1'
i132            '1'
i133            '1'
I140            '1'
I141            '1'
I142            '1'
i143            '1'
#
```

*Bibliography*


1. Cohen, Kenneth Bruce. *Model-Based Reasoning in Electronic Repair*. MS thesis, AFIT/GCE/ENG/90D-08, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A230503).
2. Cohen, Norman H. *Ada as a Second Language*. New York: McGraw-Hill Book Company, 1986.
3. Comeau, Ronald C. *Parallel Implementation of VHDL Simulations on the Intel iPSC/2 Hypercube*. MS thesis, AFIT/GCE/ENG/91D-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A243760).
4. Davis, Randall. "Diagnostic Reasoning Based on Structure and Behavior," *Artifical Intelligence*, *24*:347–410 (December 1984).
5. Department of Defense. *Requirement 64 - Microelectronic Devices*. MIL-STD 454L. Washington: DOD, 10 September 1987.
6. Dries, Flt Lt Walph W. *Model-Based Reasoning in the Detection of Satellite Anomalies*. MS thesis, AFIT/GSO/ENG/90D-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A230535).
7. Hamscher, Walter. "Modeling Digital Circuits for Troubleshooting: An Overview." *Proceedings of the IEEE 6th Conference on Artificial Intelligence Applications*. 2–8. New York: IEEE Press, 1990.
8. Lin, Dekang and Randy Goebel. "A Minimal Connection Model of Abductive Diagnostic Reasoning." *Proceedings of the IEEE 6th Conference on Artificial Intelligence Applications*. 16–22. New York: IEEE Press, 1990.
9. Ng, Hwee Tou. "Model-Based, Multiple Fault Diagnosis of Time-Varying, Continuous Physical Devices." *Proceedings of the IEEE 6th Conference on Artificial Intelligence Applications*. 9–15. New York: IEEE Press, 1990.
10. Perry, Douglas L. *VHDL*. New York: McGraw-Hill, 1991.
11. Randall Davis, Walter C. Hamscher. "Model-Based Reasoning: Troubleshooting." *AI at MIT 1*, edited by Sarah A Shellard Patrick H Winston, Cambridge, Mass: MIT Press, 1990.
12. Roger Lipsett, Carl F. Schaefer, Cary Ussery. *VHDL: Hardware Description And Design*. Boston: Kluwer Academic Press, 1991.
13. Scarl, E A., et al. "Diagnosis and Sensor Validation Through Knowledge of Structure and Function," *IEEE Transactions on Systems, Man, and Cybernetics*, *17*:360–369 (May 1987).
14. Skinner, James M. *A Diagnostic System Blending Deep and Shallow Reasoning*. MS thesis, AFIT/GCE/ENG/88D-5, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988 (AD-A202547).
15. Stroustrup, Bjarne. *The C++ Programming Language, second edition*. New York: Addison-Wesley, 1991.

## *Vita*

David R. Griffin was born in Fort Sam Houston, Texas on August 21, 1965. He graduated from Brookhaven High School, Brookhaven, Mississippi in May, 1983. He attended Mississippi State University on a National Merit and ROTC scholarships. He completed a Bachelor of Science in Electrical Engineering and Computer Engineering. While attending Mississippi State, he joined Tau Beta Pi. Upon graduation, he was commissioned a second lieutenant in the United States Air Force. Awaiting his first active-duty assignment, he spent 9 months at the Waterways Experiment Station in Vicksburg, Mississippi. He was assigned to the 3302 System Support Activity at Keesler AFB, Mississippi. He was responsible for the developement of Merlin, a computer based instructional system. In 1990 he was granted admission into the School of Engineering, Air Force Institue of Technology at Wright Patterson AFB, Ohio.

Permanent address: Rt 2 Box 286 B
Bogue Chitto, MS 39629

# REPORT DOCUMENTATION PAGE

| | | |
|---|---|---|
| | December 1992 | Master's Thesis |

A VHDL Interpreter for Model-Based Diagnoses

David R. Griffin, Captain, USAF

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GCE/ENG/92D-03

Approved for public release; distribution unlimited

Model-based reasoning permits diagnostic applications to be written without waiting for someone to become an "expert" of the system. For model-based diagnostics, there must be a model to reason from. This thesis explores using a VHDL description of the system as that model. A system based around a VHDL interpreter was written specifically for a model-based diagnostic algorithm. Currently, the diagnostic system uses an algorithm by Dries. This algorithm was derived from Scarl's Full Consistency Algorithm. The system was designed to be modular so that different diagnostic techniques could be implemented. It is divided into three parts: a VHDL parser, a VHDL interpreter, and a set of routines to implement Dries' Diagnose algorithm. The system can find stuck-at faults on combinatorial digital circuits.

VHDL, Diagnostics, Artificial Intelligence

322

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

Standard Form 298 (Rev. 2-89)